ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

Corso di Laurea Specialistica in Informatica

Progettazione e sviluppo di un sistema di comunicazione Publish/Subscribe con routing content-based di contenuti XML

Tesi di Laurea in Sistemi Middleware

Relatore:
Chiar.mo Prof.
FABIO PANZIERI

Presentata da:
ROBERTO SANTI

Correlatore:
Dr. NICOLA MEZZETTI

I Sessione

2010/2011

Il prezzo della libertá
é l'eterna vigilanza
Karl Popper

Indice

1 Introduzione **1**

2 Stato dell'arte **3**

2.1 Message Oriented Middleware 4

2.1.1 Message Flow Graph 5

2.1.2 Tematiche aperte 7

2.2 Sistemi Publish Subscribe 9

2.2.1 Disaccoppiamento 10

2.2.2 Tipologie 11

2.2.3 Sistemi Event-Based 13

2.2.4 Sistemi Content-Based 15

2.3 Casi di studio Rileventi 16

2.4 SIENA 17

2.4.1 Concetti base 17

2.4.2 Topologia delle reti 18

2.4.3 Strategie di routing 19

2.4.4 Analisi semantica delle operazioni 21

2.5 HERMES 24

2.5.1 Design 24

2.5.2 Network 25

2.5.3 Routing 25

2.5.4 Semantica 27

2.6 HERALD 28

2.6.1 Obiettivi e concetti fondamentali 28
2.6.2 Design 29
2.6.3 Logica di sistema 30

3 Progettazione **33**
3.1 Semantica 34
3.2 Banca Dati 42
3.2.1 Logging 44
3.2.2 Storing 44
3.2.3 Forwarding 48
3.3 Componenti e Struttura 48
3.4 Robustezza verso attacchi di tipo Denial of Service 51

4 Sviluppo **55**
4.1 Protocollo di Comunicazione XML 55
4.2 Piattaforma di Sviluppo e Componenti 56
4.2.1 Service Engine 58
4.2.2 Binding Component 64
4.2.3 Shared Library 66
4.2.4 Service Assembly 67
4.3 Logica e Persistenza 71
4.3.1 Tabelle Relazioni ed Entity Bean 72
4.3.2 Logica e Session Beans 75

5 Test e Performance **81**
5.1 Configurazione dei Test 81
5.2 Preparazione all'attività di testing 82
5.3 Andamento del throughput al variare delle dimensioni del messaggio 84
5.4 Andamento del throughput al variare dei subscriber 87
5.5 Andamento del throughput al variare del carico, in condizioni di stress 91

Conclusioni e Sviluppi Futuri **98**

Bibliografia **99**

Elenco delle figure

3.1 *Diagramma delle Entità Fondamentali* 43
3.2 *Schema ER Bancadati* 45
3.3 *Architettura del sistema* 49

4.1 *Architettura Open ESB* 57
4.2 *Struttura del componente* 59
4.3 *Message Exchange Pattern InOut* 63
4.4 *JBI Message Normalized Router* 63
4.5 *Funzionamento di un Binding Component* 65
4.6 *Struttura del Service Assembly* 67

5.1 *Andamento del throughput con contenuti di 1.66 Kb* 84
5.2 *Andamento del throughput con contenuti di 3 Kb* 85
5.3 *Andamento del throughput con contenuti di 5 Kb* 85
5.4 *Andamento del throughput con contenuti di 8 Kb* 86
5.5 *Andamento del throughput con contenuti di 10 Kb* 86
5.6 *Andamento del throughput al variare delle dimensioni dei messaggi* 87
5.7 *Andamento del throughput con 1 subscriber* 88
5.8 *Andamento del throughput con 2 subscriber* 89
5.9 *Andamento del throughput con 4 subscriber* 89
5.10 *Andamento del throughput con 8 subscriber* 90
5.11 *Andamento del throughput con 16 subscriber* 90
5.12 *Andamento del throughput al variare del numero di subscriber* 91

5.13 *Andamento del throughput con 144% del carico* 92
5.14 *Andamento del throughput con 200% del carico* 92
5.15 *Andamento del throughput con 244% del carico* 93
5.16 *Andamento del throughput con 300% del carico* 93
5.17 *Andamento del throughput con 333% del carico* 94
5.18 *Time to crash al variare del carico* 95

Elenco dei listati

3.1 *Listato advertise.xml* 39
3.2 *Listato subscribe.xml* 40
3.3 *Listato publish.xml* 41
3.4 *Listato Ack.xml* 42
3.5 *Query per determinare le azioni di instradamento* 47
4.1 *Listato jbi.xml del componente RingOneEngine* 60
4.2 *Listato message.java* 61
4.3 *Listato jbi.xml per la shared library* 66
4.4 *Listato jbi.xml del service assembly* 67
4.5 *Listato fileBindingINSU.wsdl* 69
4.6 *Listato Namespace.java* 72
4.7 *Listato persistence.xml* 73
4.8 *Listato della procedura StoreNotify* 76
5.1 *Configurazione parametri HTTP* 81

Elenco delle tabelle

4.1 *Mappatura delle operazioni di sistema* 56
4.2 *Elenco delle tabelle della banca dati* 72

Capitolo 1

Introduzione

Molti contesti industriali sono caratterizzati da molteplici attori che costruiscono le proprie attività di business sullo scambio di informazioni che possono essere cataloghi, listini, offerte, ordini, etc. Sfortunatamente, la gran parte dei contesti industriali è nata prima dell'era dell'informazione e ha vissuto l'informatizzazione dei processi industriali senza che si creasse l'occasione di ragionare su un'insieme di standard e di tecnologie che, adottate in maniera congiunta dai diversi player dello stesso contesto industriale, potessero massimizzare il ritorno degli investimenti attraverso una forte ottimizzazione nell'implementazione dei processi. Tale mancanza di coordinamento ha portato alla situazione attuale, ove molteplici attori devono tuttora investire rilevanti quantità di denaro per creare nuove basi informative che abilitino ulteriori possibilità di business.

In questo contesto si sente la necessità di un sistema di integrazione che operi da intermediario per lo scambio di dati tra un numero arbitrario di partecipanti, ciascuno dei quali può essere produttore o consumatore di contenuti, e che soddisfi i seguenti requisiti

- **Multucanalità**: I servizi messi a disposizione dal sistema devono essere fruibili attraverso diversi protocolli di trasporto;
- **Scalabilità**: La configurazione delle politiche di instradamento dei contenuti deve scalare all'aumentare degli attori che interagiscono col

sistema;

- **Flessibilità**:
  - il sistema deve essere integrabile con moduli o componenti che arricchiscano la logica di elaborazione dei contenuti trasmessi e mantenuti;
  - il sistema deve poter gestire contenuti di formato diverso; le regole di instradamento devono poter essere descritte attraverso gli attributi del formato di contenuto specifico;
- **Persistenza**:
  - il servizio deve mantenere lo storico delle operazioni effettuate;
  - il servizio deve mantenere una base di conoscenza aggiornata con le versioni più recenti dei contenuti pubblicati;
- **Performance**: il sistema deve erogare i propri servizi in maniera performante a molteplici attori simultaneamente;
- **Robustezza**:
  - il sistema deve essere in grado di tollerare carichi superiori alla propria capacità di servizio per periodi limitati e senza subire guasti di tipo crash;
  - il sistema deve prevedere degli accorgimenti che rendano possibile scongiurare il successo di alcune tipologie di attacco di tipo denial of service.

Con lo scopo di creare una base tecnologica sulla quale costruire un sistema che possa essere impiegato per risolvere i problemi di scambio di dati che abbiamo introdotto precedentemente, in questa tesi ci poniamo il problema di progettare e sviluppare un sistema di comunicazione di contenuti con paradigma publish/subscribe, che implementi un modello di instradamento di tipo content-based, e che soddisfi i requisiti che abbiamo elencato sopra.

Capitolo 2

Stato dell'arte

Introduzione

Il capitolo inizia da una breve introduzione sui sistemi middleware orientati ai messaggi. Da questi si scenderà nel dettaglio andando ad analizzare i sistemi Publish/Subscribe come evoluzione ed astrazione del modello MOM.

Analizzeremo, in dettaglio, le caratteristiche salienti quali il disaccoppiamento e le varie categorie. Particolare enfasi sarà concessa ai sistemi orientati agli eventi e a quelli content based.

Infine, dopo aver fatto una panoramica generale sull'argomento, andremo ad analizzare tre importanti casi di studio: Siena, Hermes ed Herald la cui analisi sarà poi utilizzata come base di partenza per il lavoro di progettazione.

I sistemi Publish/Subscribe sono stati sviluppati per poter consentire la comunicazione tra diverse entità in maniera completamente disaccoppiata, senza troppo interferire con il funzionamento e la struttura dei sistemi stessi. Attualmente questo tipo di tecnologia è largamente usata nei processi di business ed in particolare nelle architetture SOA e B2B.

Grazie alle loro caratteristiche tali sistemi sono stati impiegati nella così detta "glue technology"; ovvero quella tecnologia di contorno che consente, a sistemi già funzionanti e da tempo schierati, di poter disporre di strumenti infrastrutturali per la comunicazione e il collegamento (da qui il termine

,glue, colla), basati sul concetto di messaggio, anche laddove, al momento della progettazione, questi non sono stati previsti.

Un sistema così realizzato permette di non intralciare l'evoluzione tecnologica delle architetture ad esso collegate, evitando di trasformarsi in un elemento di appesantimento.

2.1 Message Oriented Middleware

Un Message Oriented Middleware, più comunemente indicato con MOM, è un'infrastruttura client/server che, distribuendo un'applicazione tra più piattaforme eterogenee, ne incrementa l'interoperabilità, la portabilità e la flessibilità [BCSS99].

Tale infrastruttura semplifica lo sviluppo di applicazioni che usano sistemi operativi e protocolli di rete diversi, permettendo al programmatore di ignorare i dettagli degli stessi.

Ciò si ottiene grazie a delle API che coprono diverse piattaforme e tipologie di rete. Il software MOM risiede sia sul lato client che server e tipicamente permette chiamate asincrone tra le applicazioni client e server.

Grazie ad apposite code, i messaggi inviati verso programmi non disponibili o non connessi vengono memorizzati per essere letti successivamente, ciò consente al programmatore una maggiore astrazione dal paradigma di comunicazione. I MOM costituiscono una categoria di software di comunicazione di applicazioni, generalmente basata su message passing asincrono, contrariamente al paradigma request-response.

Come avviene nei sistemi di Publish/Subscribe anche nelle architetture MOM i Client si registrano nella veste o di Publisher o Subscriber di messaggi. I messaggi vengono inviati e recapitatati a partire da un *Information Space* ognuno dei quali possiede un determinato formato per i messaggi detto *Message Schema*.

La propagazione da un spazio ad un altro avviene mediante un *Message Flow Graphs* che serve a mappare le modalità con le quali un messaggio viene

modificato ed inoltrato attraverso vari Information Space.

Il Message Flow Graphs inoltre può anche eseguire operazioni di routing filtrato verso determinati spazi, oppure eseguire delle operazioni di modifica e riadattamento dei messaggi stessi.

Per quanto riguarda invece gli Information Space questi possono mantenere delle informazioni sullo storico dei messaggi in entrata e in uscita anche di altri spazi.

Questi messaggi saranno poi utilizzati in varie modalità e per vari scopi consentendo anche che un'applicazione non sia costantemente connessa per poter ricevere messaggi (disaccoppiamento temporale).

Nel modelllo MOM, comunque, parecchie questioni rimangono irrisolte mentre non tutte quelle risolte sono implementabili in maniera efficiente.

Infatti un modello che fornisca una soluzione efficace ed efficiente per tutte le problematiche che incorrono nello sviluppo di sistemi MOM e più in generale per la "glue technolgy", non è ancora stato realizzato. Anche se parecchie limitazioni sono state superate grazie all'avvento dei sistemi Publish Subscribe. Molte questioni di importanza cruciale rimangono aperte, come ad esempio il problema della scalabilità, il problema della sicurezza nell'inoltro dei messaggi e altre questioni di tipo tecnologico. Altre importanti tematiche che riguardano il fault-tollerance, l'ordinamento dei messaggi ed altre affrontate e risolte in altri contesti, nel modello MOM non hanno ancora una soluzione efficace.

2.1.1 Message Flow Graph

I sistemi MOM, in generale, si basano sui concetti di Information Space e Message Flow.

Il sistema viene modellato come un Message Flow Graphs, dove quest'ultimo diventa una rappresentazione astratta della propagazione dei messaggi, completamente separato dai modelli di topologia di rete.

Il Message Flow Graphs [BCSS99] consiste in un grafo diretto aciclico i cui nodi rappresentano gli Information Space mentre gli archi altro non sono

che un'astrazione del flusso dei messaggi. Ogni spazio di informazione ha uno schema che specifica la struttura dei messaggi o dello stato.

I Publishers sono le sorgenti dei messaggi mentre i Subscribers sono la destinazione; quindi il flusso rappresenta la direzione di propagazione dei messaggi nel sistema. Inoltre è necessario che debbano essere mantenute delle relazioni tra i vari Information Space e i nuovi messaggi che vengono di volta in volta aggiunti al sistema. In particolare queste relazioni sono:

- **Selection**: il destinatario dei messaggi riceve un sottoinsieme di messaggi prodotti dalla sorgente che soddisfano un particolare predicato booleano P. Generalmente per predicato si intende ciò che il Subscriber ha specificato all'atto della sua sottoscrizione.

- **Transform**: i messaggi che la destinazione riceve non sono generalmente gli stessi che la sorgente ha prodotto infatti vengono eseguite su di essi delle trasformazioni necessarie ad adattarli ad eventuali formati differenti

- **Merge**: quando ci sono delle destinazioni che ricevono numerosi messaggi da più sorgenti, ovvero quando nel grafo ci sono più archi entranti sullo stesso nodo, la destinazione riceve un merge dei messaggi (in un ordine non deterministico) delle varie sorgenti. Questa operazione viene eseguita su tutti quei sistemi per i quali esistono più Publisher diversi.

- **Collapse**: lo stato dei destinatari viene computato tramite delle funzioni di riepilogo che calcolano delle particolari richieste nell'ambito dello spazio di informazione della sorgente. (Per esempio il destinatario può essere interessato a dei particolari sottoinsiemi di messaggi ricevuti e non a tutti quelli che gli sono stati recapitati)

- **Expand**: è il concetto opposto a quello di collapse: i messaggi per un determinato destinatario sono considerati tutti equivalenti e quindi non vengono fatte distinzioni e operazioni di riepilogo. Tutti i messaggi vengono inviati al destinatario in un ordine non deterministico.

Il Message Flow Graphs può evolvere dinamicamente e i suoi cambiamenti vengono considerati dei meta eventi per i quali si può fare una sottoscrizione.

Come in tutti i sistemi di questo tipo esistono delle politiche di sicurezza che distribuiscono permessi ai processi che possono eliminare dei nodi, degli archi oppure creane di nuovi. Se si osserva bene il modello appena descritto si può notare come siano evidenti le somiglianze con un sistema DBMS di tipo relazionale.

Innanzi tutto gli Information Space sono estremamente simili alle tabelle del modello relazionale infatti come quest'ultime, che hanno un proprio schema, gli Information Space ne hanno uno per il formato dei messaggi.

Altra caratteristica comune è l'operazione di select che per entrambi consiste nel selezionare un sottoinsieme di messaggi o tuple.

In ultimo è anche da considerare la funzione di collegamento svolta dagli archi del Message Flow Graphs che è del tutto simile alle chiavi esterne tra tabelle.

2.1.2 Tematiche aperte

Il modello MOM nonostante sia estremamente semplice e si basi su concetti solidi, ha alcune lacune che tutt'ora non sono state colmate e per questo non viene considerata una soluzione percorribile nelle attuali implementazioni di sistemi middleware.

Tali questioni aperte riguardano diversi ambiti che toccano tanti importanti aspetti dell'intero modello. Le prime riguardano il modello stesso: in particolare l'aspetto riguardante il linguaggio da utilizzare nella realizzazione del predicato P, il controllo degli accessi e la definizione in maniera corretta della struttura e dell'organizzazione dello spazio di informazione.

Per quest'ultima questione, che risulta essere la più importante, il problema fondamentale è come rendere la rappresentazione di uno spazio di informazione in maniera razionale e congruente, come avviene nel modello relazionale dei data base, ai quali il modello MOM in un certo senso si ispira. Infatti sarebbe estremamente difficile separare i vari tipi all'interno di Infor-

mation Space senza perdere di consistenza e senza perdere le caratteristiche intrinseche dello spazio stesso.

Una questione che, di importanza cruciale, attualmente è stata risolta è il problema della scalabiltà e in particolare la questione anche detta del message matching [BCSS99].

Il message matching consiste nel trovare degli algoritmi efficienti e sicuri che permettano di confrontare in maniera veloce un ristretto numero di eventi con un numero di possibili sottoscrizioni estremamente più elevato (anche di alcuni ordini di grandezza). Se per esempio si ha un sistema con un numero di sottoscrizioni N ,(che può superare anche 10.000) ci si aspetterà che al massimo, un numero estremamente ridotto di eventi, corrispondano esattamente con alcune sottoscrizioni questo numero viene anche detto K.

Gli algoritmi esistenti utilizzando un meccanismo di lookup con tabelle, riescono ad avere delle prestazioni in tempo costante e quindi ottimo. Grazie a questi algoritmi il message matching non è più un problema per i sistemi orientati ai messaggi. Questo tipo di algoritmi non funzionano in ambito di sistemi di tipo content-based poiché le sottoscrizioni possono fare riferimento a diversi campi dello schema di un messaggio.

Un aspetto molto importante da considerare parlando del problema del matching è che il grafo è implementato generalmente su una rete di server, distribuita geograficamente, chiamati message broker. I message broker devono quindi eseguire le funzioni di routing e multicasting ma anche le operazioni di selezione e di trasformazione.

Per andare incontro a questa problematica si adottano generalmente due soluzioni.

La prima soluzione consiste nel far eseguire le operazioni di *message matching ai Subscriber*, le informazioni così ottenute verranno poi usate per fare il routing.

Questa soluzione comunque ha un notevole elenco di svantaggi; infatti non può essere utilizzato il routing diretto nel caso di un elevato numero di client, poiché il routing point-to-point non porta vantaggi

apprezzabili nel caso di cammini comuni. Il routing basato su liste di destinazione risulta anch'esso non applicabile perché potrebbe comportare la crescita a dismisura dell'intestazione dei messaggi. Il routing basato sul multicast non porterebbe significativi vantaggi poiché che richiederebbe un elevato ed improponibile numero di gruppi.

Un'altra soluzione, anch'essa con non pochi svantaggi, consiste nell'inviare in broadcast i *messaggi a tutti i message broker* in modo tale che poi ogni singolo server faccia le operazioni di matching e poi diffonda i messaggi ai client ad esso connessi.

Lo svantaggio più grande consiste nel fatto che se le sottoscrizioni sono numerose e selettive, alcuni broker rischieranno di essere sommersi dai messaggi senza che nessuno dei client ad esso connesso necessiti di un messaggio.

In conclusione possiamo dire che l'utilizzo dei sistemi orientati ai messaggi porta dei vantaggi a livello di semplicità, di implementazione e gestione delle risorse e della rete, ma il modello su cui si basano porta con se ancora notevoli problematiche di funzionamento a livello di scalabilità e prestazioni. Questi sistemi comunque sono la base su cui si fondano i sistemi Publish/-Subscribe che in un certo senso hanno soppiantato i vecchi Message Oriented Middleware, e che attualmente ne incarnano l'evoluzione.

2.2 Sistemi Publish Subscribe

I Message Oriented Middleware forniscono una piattaforma che consente di avere una gestione dei sistemi distribuiti molto efficiente grazie al paradigma di comunicazione asincrono. Tale paradigma permette di creare una sorta di separazione tra il sistema stesso e i soggetti esterni che ne richiedono i servizi.

Basandosi su questo principio, sono stati sviluppati dei sistemi, detti Publish Subscribe, che forniscono un disaccoppiamento totale tra le varie entità che vi fanno riferimento.

La categoria dei *Publisher* raccoglie tutti quei servizi e soggetti che pubblicano un messaggio o un evento sul sistema mentre la categoria dei *Subscriber* raggruppa tutte le entità esterne che richiedono di ricevere gli eventi pubblicati dai Publisher. Questo tipo di sistemi fanno del disaccoppiamento la loro caratteristica centrale creando quindi una sorta di specializzazione e raffinamento dei sistemi MOM.

Inoltre viene inserita anche l'astrazione dell'evento che va a sostituire il concetto di messaggio come oggetto di scambio tra i vari componenti.

A seguire in questa sezione andremo a spiegare in dettaglio il concetto di disaccoppiamento, dopodiché affronteremo le varie tipologie di Publish/Subscribe soffermandoci su quelle che attualmente suscitano maggior interesse: la tipologia dei sistemi event based e la loro specializzazione content based.

In ultimo andremo ad analizzare tre sistemi Publish/Subscribe orientati agli eventi: Herald , Siena e Hermes. Gli ultimi due oltre ad essere orientati agli eventi implementano anche funzioni di routing basato sul contenuto.

2.2.1 Disaccoppiamento

Il disaccoppiamento viene definito come il grado di indipendenza dei moduli all'interno di un'architettura. In una buona progettazione, lo scopo è quello di massimizzare il disaccoppiamento, così da rendere facile la sostituzione dei moduli stessi durante la fase di manutenzione. In uno scenario di comunicazione il disaccoppiamento consente di rendere indipendenti gli attori che intervengono nello scambio. Il concetto di disaccoppiamento delle parti che può differenziarsi in tre diverse direzioni: [EFGK03]

- **Spazio**: le parti che interagiscono non hanno bisogno di conoscersi a vicenda. Generalmente l'interazione avviene tramite l'Event Service e non serve che il Publisher sappia quanti Subscriber ci sono e non ha

bisogno di avere informazioni sulla natura di quest'ultimi. Dall'altro lato la situazione è la medesima anche per il Subscriber.

- **Tempo**: i partecipanti per comunicare non hanno bisogno di essere attivi nello stesso momento. Infatti il Publisher può pubblicare anche quando il Subscriber è disconnesso e allo stesso modo il Subscriber può ricevere notifiche anche dopo che il Publisher non sia più raggiungibile.

- **Flusso**: non c'è la necessità di sincronizzazione infatti la produzione e il consumo degli eventi non avvengono nel flusso di controllo principale del Publisher e del Subscriber.

Il concetto di disaccoppiamento, in ogni sua direzione, permette di rendere la comunicazione un evento meno pesante e vincolante per i processi che vi intervengono. Infatti gli impedimenti, talvolta anche gravosi, della comunicazione (sincronizzazione) vengono aggirati grazie alla reciproca estraneità degli attori.

2.2.2 Tipologie

I sistemi Publisher/Subscriber possono essere classificati in base allo schema di funzionamento sul quale si basano. Questi schemi sono tre ed hanno le seguenti caratteristiche: [EFGK03]

- **Topic-based**: gli antenati di tutti i sistemi Publish/Subscribe si basano sul concetto di *topics*. Ovvero un determinato argomento diventa la parola chiave tramite la quale i Subscriber sottoscrivono un certo pattern di eventi.

  I Publisher notificano tali eventi. Questa modalità è fortemente intrecciata con il concetto di gruppo di comunicazione. Infatti, nei primi sistemi Publish/Subscribe, sottoscrivere un pattern di eventi consisteva nell'entrare a far parte di un gruppo di comunicazione.

Nella pratica il concetto di topics ha portato alla creazione di un tipo di astrazione di programmazione che considera ogni topic come un Event Service a se stante che dispone di una propria interfaccia .

Un'importante modifica consiste nell'orchestrazione gerarchica degli argomenti.

Un sistema basato su gruppi rappresenta una suddivisione disconnessa dello spazio degli eventi, su questo spazio di indirizzamento piatto viene costruita una gerarchia che permette ai programmatori di utilizzare interfacce per la gestione delle relazioni tra eventi diversi.

Una sottoscrizione può essere fatta su qualsiasi nodo della gerarchia, con l'automatica sottoscrizione a tutti i nodi figli fino alle foglie.

Tale gerarchia può essere rappresentata in una notazione simile a quella degli URL.

- **Content-based**: classificano gli eventi non in base ad un topic (per esempio il nome) ma in funzione ad altri aspetti che possono essere i metadati associati all'evento oppure la struttura interna dell'evento stesso.

  I subscribers specificano delle coppie nome valore che poi vengono combinate per costruire un *subscription pattern* complesso. Questo pattern viene poi analizzato da un filtro che permette di associarlo agli eventi che gli corrispondono.
  I pattern vengono rappresentati in diverse maniere, per poter essere utilizzati come argomento dell'operazione di sottoscrizione. Le varie rappresentazioni sono:[EFGK03]

  Stringhe: è la rappresentazione più frequente. I pattern vengono rappresentati con stringhe che rispettano determinati formati, come ad esempio SQL o altri. Le stringhe sono poi parsate dall'apposito motore.

  Template Object: quando un subscriber sottoscrive un determinato pattern viene creato un oggetto `t`.

Per tutti gli eventi conformi a quell'oggetto viene emessa una notifica al Subscriber. Per controllare l'eventuale conformità si ricercano quali attributi dell'oggetto e dell'evento da notificare fanno matching.

Codice Eseguibile: i subscribers forniscono un predicato che permette di filtrare a runtime gli eventi. I vari Subscriber specificano le coppie nome valore, e tramite queste coppie vengono poi filtrati i vari eventi.

Un approccio di questo tipo permette di mantenere l'incapsulamento degli oggetti *notifiche* e assicura una tipazione forte in grado di ridurre la ridondanza delle query sulle notifiche.

- **Type-based**: questo schema raggruppa gli eventi non solo in base al contenuto ma anche in base alla loro struttura interna. Questa idea si differenzia dall'approccio usato nello schema Topic-Based poiché va anche ad analizzare internamente la struttura dell'evento.

  Tale modalità permette di poter eseguire il controllo sul tipo di evento a tempo di compilazione ed evitare quindi i controlli dinamici a run time.

  Questo comporta che il tipo di un evento diventi un attributo implicito e non una caratteristica dinamica come avveniva precedentemente.

Dopo questo esame iniziale che ci permette di avere un'idea generale dei sistemi Publisher/Subscriber si può passare ad analizzare gli aspetti più importanti sullo stato dell'arte.

2.2.3 Sistemi Event-Based

Gli eventi sono le unità di base su cui si fonda, come dice il nome, questo tipo di sistemi [CNF98] [ZS01].

Il fatto di poter ragionare ed operare su eventi e non più su messaggi permette di astrarre dalle caratteristiche tecniche ed implementative dei

sistemi. Lo schema tipico di interazione consiste nella presenza di client, suddivisi nelle categorie di publisher e subscriber.

I primi, producono eventi inoltrandoli al sistema e gli altri consumano tali eventi dopo che il sistema abbia provveduto ad inoltrarli. La nozione centrale di evento altro non è che la struttura con cui vengono codificati i dati e serve a rimpiazzare la meno efficiente ed espressiva nozione di messaggio.

Gli eventi vengono strutturati in modi estremamente diversi in base alle varie implementazioni, ma hanno tutti la caratteristica di poter essere trasportati a livello di rete logica come messaggi. Possiamo, quindi, considerare questi sistemi come una specializzazione ed astrazione del modello orientato ai messaggi.

La nozione di evento permette di introdurre una serie di semplificazioni sia lato Publisher che lato Subscriber mantenendo nascosta l'implementazione di meccanismi di più basso livello come ad esempio la gestione dei carichi di rete, la scalabilità ed altri fattori legati al middleware sottostante.

Lato publisher è possibile eseguire un'operazione di advertisement, che consiste nel dichiarare un'intenzione di pubblicazione, specificando un filtro sugli eventi pubblicabili. Questo implica che si possono associare pulisher ed eventi creando quindi una sorta di tabella da usare per il routing, come si vedrà nel paragrafo successivo. I filtri quindi permettono di individuare un set di eventi per i quali i subscriber possono richiedere la notifica al momento della loro pubblicazione sul sistema. In alcune implementazioni di sistemi orientati agli eventi è possibile avere un'operazione di unadvertisement che consente al publisher di ritrattare la propria dichiarazione di pubblicazione rimuovendo il filtro dal sistema.

Lato subscriber l'adozione della nozione di evento consente di poter specificare un pattern e quindi ricevere notifiche solo per quegli eventi che rispettano il pattern specificato.

La nozione di pattern [ZS01], come nel caso di filtro, viene modellata sul concetto di evento e quindi dipende strettamente dalla sua rappresentazione.

Il pattern nel dettaglio consiste in una combinazione di filtri che generano

una particolare istanza di una classe di eventi con determinate caratteristiche di forma, struttura e contenuto. Il pattern fa parte del corpo dell'operazione di subscription e rappresenta il modello di evento che una determinata notifica deve avere per poter essere inoltrata al subscriber in questione.

Come nel caso dell'operazione di advertisement anche in questo caso esiste un'operazione di unsubscription che consente al subscriber di rimuovere il pattern specificato e quindi non ricevere più le notifiche. Come vedremo successivamente non tutti i sistemi implementano queste due operazioni.

Come andremo ad osservare in seguito i sistemi che inoltrano i messaggi in base al loro contenuto (content-based)[CW02] sfruttano il concetto di pattern e filtro per controllare se una data notifica possa interessare o meno una sottoscrizione.

Nel caso invece dei sistemi event based che non controllano l'evento, l'invio può essere effettuato sulla base dell'analisi di un solo filtro specificato dai client.

I vantaggi introdotti con l'utilizzo di questa astrazione sono molteplici sia a livello di entità esterne che di sistema. Infatti le implementazioni di queste piattaforme espongono delle API che consentono di sviluppare client e server rimanendo ad un livello di astrazione molto più alto rispetto ai sistemi orientati ai messaggi visti in precedenza.

Infatti, senza bisogno di dover andare a gestire la mappatura degli eventi dei filtri e dei pattern, è possibile utilizzare le interfacce di gestione, orientate agli eventi, astraendo non solo dalla logica dei messaggi ma anche dalla logica delle implementazioni di rete.

Di seguito si introdurranno i sistemi content based che possono essere considerati come un'ulteriore specializzazione di sistemi Publish/Subscribe orientati agli eventi.

2.2.4 Sistemi Content-Based

I sistemi Publish Subscribe Content-Based[CW02] sono una specializzazione dei sistemi orientati agli eventi. Anche questi ultimi infatti si basano

sul concetto di evento ma al momento di effettuare una notifica utilizzano degli algoritmi per individuare un sottoinsieme di sottoscrizioni che corrispondono all'evento da notificare e, in base a queste informazioni, svolgono le operazioni di inoltro.

L'approccio generico si basa su una cernita delle subscription e su una loro organizzazione in una struttura parallela di ricerca (o PST), in cui ogni sottoscrizione corrisponde ad un percorso dalla radice ad una foglia dell'albero di routing.

L'operazione di abbinamento viene eseguita seguendo tutti i possibili cammini dalla radice alle foglie che hanno specificato un pattern dell'evento.

Ogni foglia è etichettata con gli identificatori di tutti i subscriber che desiderano ricevere gli eventi corrispondenti al predicato, vale a dire, tutte le prove dalla radice alla foglia effettuate per ogni nodo intermedio.

Una volta che la struttura dati è stata modellata è possibile iniziare a definire l'algoritmo di routing degli eventi [ACW01].

Il funzionamento dell'algoritmo consiste nel discendere l'albero e nel caso in cui si raggiunge la foglia senza mai incontrare un controllo che restituisca un `false`, l'evento viene inviato al subscrber indicato dall'etichetta della foglia.

La funzione principale del sistema Content-Based consiste nel confrontare il filtro specificato dal publisher nell'advertisement con i pattern proposti dai subscriber. Dopo questo confronto è possibile creare un ristretto gruppo di advertisement che possono soddisfare la sottoscrizione. Quando un publisher pubblicherà un evento il sistema individua il sottoinsieme di sottoscrizioni e controlla quali, per quell'istanza di evento, hanno una corrispondenza.

2.3 Casi di studio Rileventi

In questa sezione analizzeremo tre tra i più importanti sistemi Publish/-Subscribe. Per ognuno di essi dopo una veloce panoramica, saranno analiz-

zate le operazioni e la semantica del sistema e soprattutto le modalità con cui questi sistemi interagiscono con Publisher e Subscriber.

2.4 SIENA

Siena è una pietra miliare nell'universo dei sistemi Publish/Subscribe content-based. La sua importanza è dovuta al fatto che esistono varie implementazioni e vari framework che consentono di poter utilizzare, anche in campo industriale e produttivo, tale sistema.

2.4.1 Concetti base

Nella progettazione del sistema viene fatta una suddivisione in due categorie degli oggetti che andranno ad interagire con *Siena*:[Car98] la categoria delle *interested party* e la categoria degli *objects of interest*.

La prima indica la categoria di consumatori di eventi che rispettino determinati pattern, la seconda categoria invece raggruppa tutti i produttori di eventi. Un sistema Publish/Subscribe deve poter gestire ovviamente le sottoscrizioni (subscription) e le pubblicazioni (notification), ma anche le sottoscrizioni degli objects of interst che vengono indicate con il concetto di advertisement. Questi tre concetti generano dei flussi di informazione che coinvolgono il sistema, le intersted party e gli objects of interst.

Nel caso dei concetti di advertisement e subscription il flusso viene generato dalle entità esterne e si muove in direzione del sistema mentre, nel caso del concetto di notification, il flusso ha un verso opposto.

Una rappresentazione informale dei flussi potrebbe essere la seguente: un objects of interst, tramite advertisement, specifica le proprie intenzioni di pubblicare eventi. Tramite subscription, un'intersted party specifica il desiderio di ricevere notifiche di eventi che rispettino il pattern proposto. Quando un evento viene pubblicato sul sistema quest'ultimo provvederà a notificare all'intersted party, che ha fatto la sottoscrizione, mediante una notification.

Infine sono molto importanti i concetti di pattern e di filter. Il patter viene specificato dal subscriber nell'operazione di sottoscrizione e si tratta di un modello strutturale individuato da coppie tipo valore: se un evento rispetta tale pattern allora potrà essere inoltrato al subscriber che ha prodotto il modello.

Il filter invece viene prodotto con un messaggio di advertisement, dal publisher, mediante il si quale individua un sottoinsieme degli eventi. Tutti gli eventi che tale publisher produrrà apparterranno a questo sottoinsieme.

Dopo questa breve introduzione sui concetti fondamentali di Siena, in seguito andremo a parlare della struttura della rete logica che i nodi del sistema formano e del concetto fondamentale di routing.

2.4.2 Topologia delle reti

Le connessioni attraverso i nodi di Siena si basano sul protocollo di IP unicast ma il sistema crea un'astrazione che permette di considerare le comunicazioni solo a livello logico di applicazione e non a livello di protocollo di rete [Car98].

In Siena si possono individuare quattro tipi di topologie di rete logica che andremo brevemente a spiegare:

- **Gerarchica**: ogni server ha delle connessioni con dei figli che possono essere indistintamente objects of interest, interested party o altri server. Ogni nodo server ha delle connessioni verso il server padre. Le connessioni verso il nodo superiore hanno sempre verso uscente.

  In questo tipo di topologia il protocollo utilizzato per le comunicazioni client-server e server-server è lo stesso, infatti tale struttura della rete è una naturale evoluzione dell'architettura client server originaria.

  Il problema maggiore dovuto dall'adozione di questa topologia è che ogni nodo diventa un punto critico sensibile. Infatti la caduta di un nodo disconnetterebbe l'intera sottorete raggiungibile attraverso il padre.

- **Aciclica Peer-to-Peer**: la comunicazione utilizza due protocolli uno per la comunicazione tra client e server e uno per la comunicazione tra server e server. I canali tra i vari server sono tutti bidirezionali e permettono quindi un flusso in entrambi i sensi di advertisement, subscription e notification. Gli algoritmi di routing che vengono adottati per l'inoltro di messaggi, sfruttano la proprietà che il grafo composto dai canali di comunicazione e dai server sia aciclico. Quando si aggiungono dei nodi la principale accortezza è quella di fare in modo di non violare tale proprietà.

  Come nel caso precedente anche utilizzando questa topologia se un nodo cade anche la sottorete ad esso connessa diventa irraggiungibile.

- **Generica Peer-to-Peer**: qui viene rimosso il vicolo del grafo aciclico e si ottengono una serie di vantaggi che vanno dalla maggiore affidabilità della rete (non esiste più il problema del singolo nodo critico) al minor carico dei canali di comunicazione.

  Per questo tipo di topologia, comunque, si richiede un algoritmo che tenga conto anche dei cammini minimi e che gli eventi inoltrati debbano avere un time-to-live.

- **Ibrida**: una topologia ibrida viene spesso realizzata per esigenze di prestazioni e di efficienza di gestione. Nella maggior parte dei casi si realizza una topologia che comprende una serie di sottoreti separate di tipo “”Generica“” che sono collegate tra loro tramite una topologia di tipo Aciclica.

2.4.3 Strategie di routing

Nelle strategie di routing Siena sfrutta il fatto che i pattern specificati nelle sottoscrizioni si possono sovrapporre e quindi in alcuni casi è possibile risparmiare alcune risorse nell'invio dei messaggi. Inoltre l'idea alla base del routing si basa su due principi fondamentali:[Car98]

- **downstream duplication**: questo principio dice che le notifiche devono essere inoltrate in singola copia, per quanto possibile e che la replica deve essere fatta, il più vicino possibile alle intersted parties che hanno sottoscritto l'evento.

  Ovvero la notifica deve essere duplicata solamente quando, da un server di routing padre, deve essere inviata a più di un server figlio. In sostanza la replica deve avvenire solamente quando è realmente necessaria.

- **upstream monitoring**: con questo principio si vuole indicare che deve essere eseguito un accorpamento dei flussi di eventi il più vicino possibile alla sorgente dell'evento o del sub-pattern. All'interno del percorso di routing dovrebbe esistere un nodo che monitorizza ed eventualmente accorpa più flussi di eventi il più vicino possibile alla sorgente del flusso stesso.

In Siena si hanno due classi di algoritmi di routing.La prima classe *subscriptions forwarding* utilizza come principio il downstream duplication e realizza un algoritmo di routing basato sulle subscription mentre l'altra classe, *advertisement forwarding* realizza un algoritmo basato sull'advertisement ed utilizza entrambi i principi elencati in precedenza.

Nel dettaglio:

- **subscriptions forwarding**: in questa classe di algoritmi, i percorsi di routing per le notifiche sono impostati dalle subscription inoltrate in broadcast a tutta la rete. Ogni subscrption viene archiviata e trasmessa dal server di origine a tutti i server delnetwork in modo da formare un albero che collega il subscriber a tutti i server.

  Quando un oggetto pubblica una notifica, che corrisponde a quella sottoscrizione, la notifica viene instradata verso il subscriber seguendo il percorso inverso fatto dalla sottoscrizione dell'evento.

- **advertisement forwarding**: questa tecnica utilizza l'advertisement per impostare i percorsi per le sottoscrizioni, che a loro volta andranno ad impostare il percorso per le notifiche.

Ogni advertisement è trasmesso in tutta la rete, in modo da formare un albero che tocca tutti i server. Quando un server riceve una sottoscrizione, la propaga nel senso contrario , lungo il percorso dell'advertisement. Le notifiche vengono quindi inoltrate solo attraverso i percorsi attivati dalle advertisement.

2.4.4 Analisi semantica delle operazioni

In questa sezione si prenderà in esame la semantica delle operazioni individuate nella sezione precedente.

Per definire in maniera precisa la semantica delle operazioni bisogna innanzi tutto introdurre e definire il concetto: di *compatibilità*.

Il concetto di compatibilità stabilisce un'importante relazione tra la subscription e la notification. Infatti il ruolo fondamentale del sistema di Publish/Subscribe, è quello di pubblicare delle notifiche a dei subscriber che facciano match con le sottoscrizioni di quest'ultimi, e quindi, tramite la compatibilità, viene stabilita la semantica dell'operazione di subscription.

La compatibilità stabilisce inoltre anche una relazione tra advertisemnt e subscription. Questa relazione risulta essere molto importante in fase di routing poiché il sistema fa un'analisi dell'advertisement per controllare se sia rilevante per qualche subscription.

In questo modo la compatibilità mette in relazione la notifica con il concetto di advertisement definendo la semantica dell'operazione di advertise.

La definizione della semantica delle operazioni permette di stabilire le modalità di risposta del sistema ad un'operazione di advertise e di subscription. Si può quindi determinare una semantica di sistema: una di tipo advertise ed una di tipo subscription: queste due diverse semantiche danno origine a due diversi tipi di sistemi Publish/Subscribe: *subscription-based* e uno *advertise-based* [Car98].

Nei sistemi di tipo subscription-based la semantica del sistema è determinata solamente dalle operazioni di subscription. In questo caso le operazioni

di advertise non sono essenziali per lo scopo ultimo del sistema; come viene definito nella relazione di compatibilità tra subscription e notification.

Da questa relazione deriva, quindi, che il sistema invia una notifica `n` ad un oggetto `X` se e solo se `X` ha effettuato un'operazione di subscription che mette in relazione l'operazione stessa e la notifica `n`. Ovviamente `X` non deve aver eseguito un'operazione di unsubscription per la sottoscrizione precedente.

Con questo tipo di semantica se la relazione non è soddisfatta al tempo della notifica l'inoltro da parte del sistema non termina e l'intersted party non riceve l'evento.

Nei sistemi che utilizzano una semantica di tipo advertise-based entrambe le operazioni di advertise e subscription sono fondamentali per gli scopi del sistema.

Il sistema invia una notifica `n`, postata da un oggetto `X`, ad un oggetto `Y` che l'ha sottoscritta, se e solo se vengono rispettate alcune condizioni.

Queste condizioni sono: Y deve aver eseguito un'operazione di advertise `a`, `X` deve aver eseguito un'operazione di subscription `s`, `a` deve essere in relazione di compatibilità con `s`, ovvero a deve avere una certa rilevanza per `s` ed infine `s` deve essere compatibile con `n`.

Se tutte queste condizioni sono rispettate allora il sistema sarà in grado di fare il delivery della notifica. In questo caso è possibile che se un oggetto `X` fa una sottoscrizione `s1` che è compatibile con una notifica `n`, ma `Y` non ha mai fatto un'advertise che sia in relazione con la subscription `s1`, il sistema non garantisce che la notifica venga inoltrata. In questo caso le condizioni per l'inoltro del messaggio debbono verificarsi al tempo della sottoscrizione di `Y`.

Come si è detto in precedenza le operazioni unadvise e unsubscription cancellano una o più corrispondenti operazioni di subscription e advertise.

Quando un'intersted party esegue un'operazione di unsubscription vengono eliminate dal sistema tutte le subscription che sono compatibili con il filtro specificato in subscription.

Abbastanza simile è la semantica della unadvertise che elimina tutte le

advertise che sono compatibili con il filtro specificato nell'operazione.

2.5 HERMES

Hermes[PB02] è un sistema middleware, distribuito orientato agli oggetti.

Di seguito si prenderanno in esame le caratteristiche salienti del sistema ed analizzeremo le differenze con altri sistemi Publish/Subscribe, con particolare attenzione ai principi di funzionamento.

2.5.1 Design

Hermes consiste di due importanti componenti: l'*Event Client* e l'*Event Broker*.

Un Event Client può essere un publisher oppure un subscriber che utilizzano i servizi forniti dal sistema per generare, notificare e pubblicare eventi. L'event Broker invece è l'implementazione distribuita delle funzionalità che i client utilizzano per le loro operazioni sugli eventi. Questo aspetto dei broker risulta essere molto vantaggioso in termini di sviluppo dei client: infatti è possibile implementare dei publisher (o subscriber) leggeri che non devono assolutamente gestire problematiche di rete o implementare funzionalità di sistema.

La funzionalità più importante di un Event Broker è quella di accettare delle sottoscrizioni e gestire le notifiche degli eventi in base agli interessi specificati dai client. Gli Event Broker sono interconnessi uno con l'altro a formare un grafo non connesso, ed implementano una rete logica per la diffusione degli eventi utilizzando il paradigma del message-passing.

Gli eventi che vengono pubblicati su un broker sono tradotti in messaggi, che verranno trasportati sulla rete logica ed inoltrati ai subscriber.

In Hermes, come in altri sistemi Publish/Subscribe, si utilizza la proprietà di advertisement che consente al publisher di presentarsi al sistema prima che inizi a produrre eventi.

Nella maggior parte dei sistemi publish subscribe l'advertisement è utilizzato per facilitare le operazioni di routing: viene creato infatti un albero che ingloba sia i subscriber che hanno fatto le sottoscrizioni, che i publisher che

hanno generato degli advertisement in questo modo il path di una notifica diventa uno dei cammini generati a partire dall'albero ottenuto.

In Hermes viene utilizzato un approccio diverso,che implica l'utilizzo di un punto di incontro tra subscription e advertisement detto nodo di *rendezvous*. Questo nodo insieme alla rete di broker viene utilizzato per la creazione dell'albero di routing.

2.5.2 Network

La rete di routing in Hermes è una rete logica a livello di applicazione.

I nodi che costituiscono questa rete, altro non sono che gli Event Broker, che fanno routing di messaggi inoltrandoli ad altri nodi.

Ogni Event Broker della rete ha un id unico e l'operazione di inoltro consiste in una chiamata che ha come parametri il messaggio stesso e l'id del nodo di destinazione. Se l'id della destinazione non esiste nella rete, il messaggio viene inviato al broker con l'id numerico più vicino.

Nell'architettura della rete esistono dei nodi particolari detti *nodi di rendezvous* che sono conosciuti sia dai Publisher che dai Subscriber. I nodi di rendezvous funzionano come dei punti di incontro tra i messaggi di avertisement e i messaggi di subscription. Per individuare un determinato rendez vous point si utilizza un particolare valore hash del tipo di evento, che individua univocamente il nodo.

Una volta che l'id del nodo è stato calcolato viene utilizzata la funzione di routing per smistare sia messaggi di advertisement che messaggi di subscription. Per evitare che questi nodi diventino dei colli di bottiglia generalmente sono replicati.

2.5.3 Routing

In Hermes si utilizza un approccio ibrido che può essere considerato una sintesi del content-based e del type based, tale approccio viene detto di tipo type-and-attribute based.

La parte di subscription avviene in due fasi: la prima consiste, da parte del subscriber, nello specificare un tipo di evento (topic) e nella seconda fase un'espressione che verrà poi utilizzata come filtro per la selezione delle notifiche. L'espressione andrà ad operare sugli attributi dell'evento.

In Herald il routing degli eventi è realizzato i sfruttando sia i tipi degli eventi che gli attributi, ed è possibile realizzare una modalità di inoltro altamente scalabile e particolarmente performante rispetto agli algoritmi adottati in altri sistemi Publish/Subscribe.

Innanzi tutto si possono individuare quattro tipi di messaggi, comuni a tutti i sistemi, su di essi verrà poi modellato l'algoritmo di routing specifico di Hermes [PB02].

- **Type Messages**: aggiunge un nuovo event type al sistema e predispone un broker a diventare un nodo rendezvous per fare storage e type cheking dell'evento.

- **Advertisement Messages**: viene utilizzato dal publisher per specificare le proprie intenzioni di pubblicazione; mediante questo tipo di messaggio viene specificato il tipo degli eventi che verranno pubblicati.

- **Subscription Messages**: sono utilizzati dai subscriber per specificare il proprio interesse verso un certo tipo di eventi. Questi messaggi insieme ai messaggi di advertisement servono a costruitre l'albero di routing degli eventi. In tali messaggi viene specifiato il tipo dell'evento e l'espressione di filtro sugli attributi.

- **Publication Messages**: i messaggi di questo tipo altro non sono che una diversa rappresentazione degli eventi che vengono inoltrati tra i vari Event Broker.

Il routing type-based funziona nel modo seguente: prima che un evento possa essere instradato, il nodo di rendezvous deve essere istituito: questo avviene mediante un messaggio di tipo "type message" inoltrato a quell'Event Broker che ha come id l'hash del tipo di evento specificato.

Una volta che il nodo di rendezvous è stato creato i publisher e subscriber possono iniziare ad inviare i loro messaggi. Immaginando una rete in cui sono presenti due publisher e due subscriber, abbiamo uno scenario per cui i publisher inviano due messaggi di advertisement `a1` e `a2` al nodo di rendezvous `R`.

Per sottoscrivere l'evento i due subscriber inviano due messaggi di subscription `s1` e `s2` sempre allo stesso nodo `R`. Durante il percorso di routing ogni nodo mantine delle informazioni di stato sul tipo di messaggio che ha inoltrato, (advertisement o subscription).

Inoltre vengono archiviati, per ogni nodo, gli id del broker che precede e l'id di quello che segue. Quando un publisher pubblica un evento `p1` questo viene inoltrato verso il nodo di rendezvous e segue il percorso del messaggio di advertisement per quel tipo di evento.

Ogni volta che il messaggio raggiunge un nodo che contiene le informazioni di un messaggio di subscription per l'evento in questione, la pubblicazione `p1` segue il percorso inverso seguito dal messaggio di susbscription fino a giungere al subscriber dell'evento.

Un aspetto importante è che i messaggi di publication non devono essere per forza inoltrati al rendezvous evitando così il rischio che il nodo diventi un collo di bottiglia per l'intera rete.

Da questa analisi risulta che il sistema dei rendezvous serve essenzialmente per generare l'albero di routing. La radice di tale albero è il nodo `R` che altro non è che il punto di incontro dei messaggi di advertisement e susbscription.

2.5.4 Semantica

Si può dare, ora, una definizione dell'interfaccia che il sistema espone per i client esterni.

Come per Siena, i concetti fondamentali risultano essere la sottoscrizione (subscription) e la dichiarazione dei publisher (advertisement). Questi due concetti anche se usati in maniera differente da Hermes anche in questo caso costituiscono la parte fondamentale del routing.

L'importanza della subscription, dal punto di vista della logica del sistema, consiste soprattutto nel creare un percorso di interesse per il subscriber all'interno della rete logica degli Event Broker. Il messaggio di subscription dal punto di vista semantico ha quindi la funzione di aggiungere un filtro sugli eventi al sistema. Questo filtro permette poi di individuare quale subscriber ha manifestato l'interesse per l'evento.

In maniera analoga la funzionalità di advertisement crea nella rete un percorso di routing per l'evento pubblicato. Infatti con questi messaggi viene specificato il percorso di un evento all'interno della rete in base suo tipo. Semanticamente la funzione di advertisement altro non è che la dichiarazione di tipo che permette quindi di creare l'associazione sottoscrizione-evento.

Come per tutti i sistemi publish subscribe la pubblicazione di un evento consiste in una trasformazione del formato dell'evento in un messaggio e nell'inoltro di quest'ultimo verso il sistema. Particolarmente interessanti sono i messaggi di tipo "type message". Con questo tipo di operazione viene "eletto"un nuovo randezvous consentendo al sistema di creare un albero di routing con radice nel randezvous stesso. Questa operazione va considerata come tipica del sistema Hermes e soprattutto interna ad esso.

Per questo non risulta interessante per la formalizzazione dell'interfaccia.

2.6 HERALD

Herald[CJT01] è un sistema publish subscribe orientato agli eventi proposto da Microsoft.

La caratteristica principale del progetto è la scalabilità sia rispetto al numero di client che rispetto al numero di eventi che vengono pubblicati contemporaneamente.

2.6.1 Obiettivi e concetti fondamentali

Come accennato sopra il focus principale nella progettazione di Herald è la scalabilità del sistema, in particolar modo rivolto alla gestione dei messaggi

e alla gestione dello stato dei nodi.

In Herald la nozione di evento viene intesa come un set di dati, prodotti da un publisher sul sistema, che quest'ultimo provvederà ad inoltrare a tutti i subscriber che hanno sottoscritto quell'evento. A differenza dei due sistemi presi in considerazione precedentemente Herald non controlla il contenuto dell'evento e quindi non fa parte di quella categoria di sistemi content-based.

Un altro aspetto fondamentale di Herald è la presenza del rendezvous point. Il rendezvous point è un'astrazione che crea un punto sul quale il publisher pubblica i suoi eventi e i subscriber inviano le loro sottoscrizioni. Il rendezvous point è anche il responsabile per la notifica degli eventi ai subscriber.

Come per gli altri due sistemi sono presenti le tre operazioni fondamentali: publish subscribe e notify. Inoltre esiste un'altra operazione che viene eseguita dai subscriber, e che in Siena ed Hermes non erano presenti, chiamata creator che comporta la creazione sul sistema di un punto di rendezvous.

2.6.2 Design

I principali criteri della progettazione sono:

- **Organizzazion Eterogeneus**: Herald è costituito da una serie di organizzazioni di macchine eterogenee che costituisco un set di domini cooperanti.

- **Scalabilità**: l'implementazione del sistema dovrebbe scalare in tutte le dimensioni compreso il numero di client e il volume di messaggi che vengono prodotti dagli stessi. Inoltre dovrebbe essere scalabile anche rispetto al numero dei punti di rendezvous e rispetto al numero dei domini.

- **Elasticità**: Herald deve essere in grado di funzionare anche in caso di numerose disconnessioni e crash dei nodi. Inoltre deve essere in grado di mantenere la propria funzionalità anche quando subisce attacchi da utenti malevoli.

- **Autogestione**: il sistema deve essere in grado di prendere decisioni autonomamente su questioni di gestione interna dei dati ma anche sulla modalità di inoltro degli eventi dai publisher verso i subscriber. Inoltre deve essere in grado di adattarsi alla disponibilità di risorse senza produrre attese eccessive per i processi.

- **Reattività**: l'inoltro dei messaggi deve essere veloce ed efficiente tale da supportare le attività di tipo human-to-human.

- **Supporto per la disconnessione**: Herald deve creare delle strutture dati, tipo code, per immagazzinare temporaneamente i messaggi per i client che non sono momentaneamente connessi. Queste strutture andranno poi svuotate quando il client sarà di nuovo connesso.

- **Funzionamento partizionato**: i publisher e i subscriber che, in caso di partizionamento della rete, si trovino in partizioni diverse devono poter continuare a comunicare ugualmente. E il delivery di un evento deve essere portato a termine attraverso le diverse partizioni.

- **Sicurezza**: si vorrebbe inserire un controllo degli accessi che permetta di far accedere solo soggetti autenticati. Tale controllo potrebbe avere degli effetti negativi sulle prestazioni di Hermes.

Dopo aver elencato e spiegato i principali criteri di design di Herald si provvederà ad affrontare l'aspetto semantico del sistema e delle operazioni che quest'ultimo fornisce ai client.

2.6.3 Logica di sistema

In Herald vengono configurati tre attori principali: publisher, subscrber e rendezvous point. Il rendezvous può essere considerato come un endpoint per i publsher e per i subscriber, che rispettivamente pubblicano e sottoscrivono eventi.

In Herald non sono previste operazioni sul contenuto del messaggio quindi l'operazione di notification consiste essenzialmente nel delivery dell'evento

verso i client che hanno sottoscritto, senza alcun filtro su quell'evento; in uno scenario del genere risultano inutili eventuali messaggi di advertisement.

Infatti la funzionalità semantica della dichiarazione del publisher era quella di creare sul sistema un filtro che stabilisse un sottoinsieme di eventi, questo sottoinsieme veniva poi utilizzato per fare routing content-based. Per quanto riguarda invece l'operazione di notification può essere considerata la tipica operazione di notifica effettuata dai sistemi Publish/Subscribe non content-based.

Quindi si può affermare che Herald rispetto ai sistemi visti in precenza (Siena ed Hermes) implementa un set minimo di operazioni, per il fatto che non implementa il routing content based e non necessita quindi di avere delle operazioni di advertisement.

Capitolo 3

Progettazione

Introduzione

In questo capitolo presenteremo i passi che abbiamo seguito nella progettazione del prototipo di servizio oggetto di questa tesi. Nella fase di progettazione abbiamo seguito un approccio ontologico, ovvero abbiamo prima progettato il modello dei dati e le interfacce, e in un secondo momento abbiamo progettato le logiche del sistema. Coerente con l'approccio usato per la progettazione presenteremo, in ordine, la progettazione delle interfacce del sistema, la semantica delle operazioni, il modello della banca dati a supporto del servizio e, infine, la struttura del prototipo in termini di moduli e componenti. Le scelte progettuali sono state guidate dai seguenti requisiti:

- **Multucanalità**: I servizi messi a disposizione dal sistema devono essere fruibili attraverso diversi protocolli di trasporto;

- **Scalabilità**: La configurazione delle politiche di instradamento dei contenuti deve scalare all'aumentare degli attori che interagiscono col sistema;

- **Flessibilità**:

  - il sistema deve essere integrabile con moduli o componenti che arricchiscano la logica di elaborazione dei contenuti trasmessi e mantenuti;
  - il sistema deve poter gestire contenuti di formato diverso; le regole di instradamento devono poter essere descritte attraverso gli attributi del formato di contenuto specifico;

- **Persistenza**:
  - il servizio deve mantenere lo storico delle operazioni effettuate;
  - il servizio deve mantenere una base di conoscenza aggiornata con le versioni più recenti dei contenuti pubblicati;

- **Performance**: il sistema deve erogare i propri servizi in maniera performante a molteplici attori simultaneamente;

- **Robustezza**:
  - il sistema deve essere in grado di tollerare carichi superiori alla propria capacità di servizio per periodi limitati e senza subire guasti di tipo crash;
  - il sistema deve prevedere degli accorgimenti che rendano possibile scongiurare il successo di alcune tipologie di attacco di tipo denial of service.

3.1 Semantica

In questa sezione descriviamo il processo di definizione delle operazioni implementate del sistema, delle rispettive interfacce, e della semantica di ogni singola operazione.

Un sistema di comunicazione publish/subscribe prevede due modalità di interazioni, a seconda della tipologia di attore che accede al servizio. Vi può essere l'attore che pubblica i contenuti, che in seguito identificheremo

col nome *Publisher*, e l'attore che è interessato a ricevere i contenuti, che in seguito identificheremo col nome *Subscriber*.

Per definire l'insieme delle operazioni che il sistema deve implementare, abbiamo preso in esame i sistemi di comunicazione publish/subscribe presentati nel capitolo precedente individuando, attraverso un esame comparativo, un nucleo di operazioni che tali sistemi offrono a ciascuna delle due tipologie di attori presentati precedentemente. Per ciascuna operazione individuata, abbiamo definito una semantica che supportasse il soddisfacimento dei requisiti specificati precedentemente e nel contempo non contraddicesse quelle delle operazioni corrispondenti sui sitemi che costituiscono lo stato dell'arte.

Le operazioni rilevanti per un publisher sono quelle che implementano le seguenti azioni:

- **Dichiarare una serie di pubblicazioni**: questa operazione consente ad un publisher di formalizzare la sua intenzione di pubblicare contenuti;

- **Pubblicare un contenuto**: questa operazione consente al publisher di pubblicare un contenuto. Tale pubblicazione andrà a buon fine se il publisher pubblica un'istanza legittima di una famiglia di contenuti che aveva precedentemente dichiarato;

- **Annullare una dichiarazione di pubblicazione**: con questa operazione un publisher può annullare una dichiarazione di pubblicazione.

Le operazioni rilevanti per un subscriber sono quelle che implementano le seguenti azioni:

- **Sottoscrizione**: tale operazione deve consentire all'attore di specificare la famiglia di contenuti che intende ricevere, nonché le condizioni che le istanze di tale famiglia devono rispettare perché l'inoltro sia implementato. Per esempio, dato uno schema xml che descrive il formato delle schede dei prodotti in commercio e dati due elementi di tale schema che identificano le informazioni sulla marca e sul fornitore dei

prodotti descritti nelle schede, allora l'operazione di sottoscrizione deve ad esempio poter consentire all'attore di sottoscriversi alla ricezione di tutte le schede prodotto che presentano specifiche combinazioni di valori per quegli attributi;

- **Modifica di una sottoscrizione**: tale operazione consente all'attore di modificare le condizioni di inoltro per una sottoscrizione effettuata precedentemente;
- **Annullamento di una sottoscrizione**: tale operazione consente all'attore di annullare una sottoscrizione effettuata precedentemente;
- **Recupero del contenuto** (qualora la modalità di recupero dei contenuti fosse di tipo polling).

Ai fini dell'implementazione del prototipo, abbiamo assunto un meccanismo di consegna dei contenuti mediante notifica e abbiamo quindi individuato le seguenti operazioni eseguibili dai rispettivi attori:

- **Publisher**
  - *Advertise*: questa operazione consente al publisher di dichiarare la sua intenzione di pubblicare contenuti appartenenti ad una famiglia specificata. Questa operazione termina con successo se il publisher è autorizzato a pubblicare contenuti appartenenti alla famiglia specificata; in caso contrario l'operazione restituirà un fallimento. Si noti che nell'implementazione prototipale non è stato implementato il meccanismo di abilitazione alle famiglie di contenuti; un publisher può decidere di pubblicare qualunque famiglia di contenuti. Quando un publisher effettua una dichiarazione per una famiglia di contenuti per cui ha già una dichiarazione non annullata, la dichiarazione termina con successo restituendo lo stesso handler della prima;
  - *Publish*: Questa operazione consente all'attore di pubblicare un contenuto. Il contenuto verrà inserito nella base di conoscenza e

inoltrato verso tutti gli attori che hanno al momento una sottoscrizione attiva che cattura il contenuto. Una publish fallisce quando il contenuto pubblicato non è un'istanza legittima della famiglia di contenuto oppure quando il publisher non ha precedentemente dichiarato che avrebbe pubblicato contenuti appartenenti a quella famiglia;

- *Unadvertise*: questa operazione consente all'attore di annullare una precedente dichiarazione, referenziata attraverso gli stessi parametri che il Publisher ha specificato nel Advertise. Questa operazione termina sempre con successo.

- **Subscriber**

  - *Subscribe*: questa operazione consente all'attore di effettuare la sottoscrizione ai contenuti, specificando anche le condizioni di inoltro per tali contenuti. Una sottoscrizione restituisce insuccesso qualora le condizioni di inoltro specificate facessero riferimento ad attributi che non sono definite nel namespace che descrive la struttura del contenuto.

  - *Unsubscribe*: questa operazione consente all'attore di annullare una precedente sottoscrizione, referenziata attraverso gli stessi parametri specificati nella sottoscrizione. Tale operazione ha sempre esito positivo.

Per quanto riguarda la specifica formale delle operazioni che abbiamo deciso di implementare, ci riferiremo allo standard web definito nella famiglia di specifiche WS-Notification, sviluppato dal consorzio OASIS (Organization for the Advancement of Structured Information Standards). In particolare, questa tesi riprende dal sottoinsieme WS-BaseNotification [SGM06] per la definizione delle interfacce dei servizi Web per i due ruoli importanti nel modello di notifica, ovvero il ruolo di NotificationProducer (che corrisponde all'attore Publisher) e il ruolo di NotificationConsumer (che corrisponde al

Subscriber). In particolare, l'obiettivo della specifica WS-BaseNotification è quello di uniformare la terminologia, i concetti, le operazioni, i WSDL e le strutture XML necessari per esprimere i ruoli fondamentali dei servizi Web di pubblicazione e sottoscrizione di notifiche. Per soddisfare tali obiettivi, la specifica WS-BaseNotification deve fare in modo che le specifiche siano implementabili anche su dispositivi con limitate capacità hardware.

Tuttavia, la limitazione che WS-BaseNotification presenta e che non consente il soddisfacimento dei nostri requisiti riguarda la maniera in cui il subscriber referenzia l'insieme dei contenuti che gli devono essere inoltrati. Secondo questo standard, un sottoscrittore specifica i contenuti che desidera ricevere attraverso la sola identificazione del publisher o l'identificazione dell'argomento o del formato.

WS-BaseNotification definisce il formato e la semantica della notifica ma non definisce le modalità con cui un Pubisher possa produrre una notifica e nemmeno la modalità attraverso la quale i subscribers vengono a conoscenza dei possibili publisher. Inoltre, WS-BaseNotification definisce che il protocollo di trasporto deve essere ortogonale alla sottoscrizione e la consegna delle notifiche, in modo che la specifica possa essere implementata su una varietà di protocolli di trasporto.

In questa tesi, il formalismo definito in WS-BaseNotification viene esteso per estendere la semantica di definizione delle sottoscrizioni e consentire la codifica delle condizioni di sottoscrizione che abbiamo descritto precedentemente.

Coerentemente con quanto definito in WS-BaseNotification, nell'implementazione del nostro prototipo le operazioni saranno esposte attraverso un web service e saranno trasportate via SOAP. I messaggi contenenti le operazioni saranno documenti XML [PSMB98] e quindi le operazioni faranno largo uso di standard e tecnologie della famiglia XML In particolare Xpath [BRC08], XML Schema [BRWF01] e Namespace [BTT+09].

Coerentemente con quanto previsto da WS-BaseNotification, la struttura del messaggio di *Advertise* consiste nella specifica di un uri che rappresenta il namespace del contenuto e un url che referenzia univocamente lo schema

XML associato a quel contenuto. Di seguito un esempio di Advertise:

```
<RegisterPublisher xmlns="http://docs.oasis-open.org/wsn/br-2"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://docs.oasis-open.org/wsn/br-2 http://docs.oasis-open.org/
        wsn/br-2.xsd">
    <PublisherReference>
        <Address xmlns="http://www.w3.org/2005/08/addressing">
            url del Publisher
        </Address>
        <ReferenceParameters xmlns="http://www.w3.org/2005/08/addressing">
            <in>url per l'invio di messaggi</in>
            <out>url per la ricezione di messaggi</out>
        </ReferenceParameters>
    </PublisherReference>
    <Topic Dialect="http://activemq.apache.org/camel/schema/spring">
        <schemalocation>
            http://societa.xmlschema.com/nome/space/ns.xsd
        </schemalocation>
    </Topic>
</RegisterPublisher>
```

Listing 3.1: `advertise.xml`

L'operazione di subscribe è eseguita da un subscriber per specificare al sistema la classe di contenuti che desidera ricevere e le condizioni che tali contenuti devono soddisfare affinché l'instradamento sia effettuato. Nella nostra implementazione, la subscription consiste nella specifica di un namespace, e di una sequenza di coppie $(condition, target)$. Per ognuna di tali coppie, *condition* è un percorso xpath. Il namespace individua la famiglia dell'evento, ovvero lo schema XML che descrive il formato del contenuto. Quando, per ogni coppia C_i, l'applicazione della $condition_i$ al contenuto C assume il valore *value*, allora il contenuto è instradato verso il subscriber.

Nella stessa operazione di subscribe, è stato inserito un attributo booleano *retroactive* che specifica se la condizione di inoltro si deve applicare anche ai contenuti che sono presenti nella base di conoscenza.

Affinché questa operazione termini con successo devono essere valide le seguenti condizioni:

- il namespace che identifica la famiglia di contenuti deve essere registrato nel sistema;
- le condizioni specificate devono essere definite all'interno di tale namespace.

Analogamente all'operazione di Advertisement, la Subscription è un'operazione idempotente; molteplici esecuzioni della stessa Subscription da parte dello stesso attore restituiscono lo stesso valore.

Il listato che segue è un esempio di Subscription.

```
<wsn-b:Subscribe xmlns:wsn-b="http://docs.oasis-open.org/wsn/b-2" xmlns:wsa="http://www
    .w3.org/2005/08/addressing" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://docs.oasis-open.org/wsn/br-2 http://docs.oasis-open.org
    /wsn/br-2.xsd">
    <wsn-b:ConsumerReference>
        <wsa:Address xmlns="http://www.w3.org/2005/08/addressing">
            url del Subscriber
        </wsa:Address>
        <ReferenceParameters xmlns="http://www.w3.org/2005/08/addressing">
            <in>url per l'invio di messaggi</in>
            <out>url per la ricezione di messaggi</out>
        </ReferenceParameters>
    </wsn-b:ConsumerReference>
    <wsn-b:Filter>
        <wsn-b:TopicExpression Dialect="namespace">
            <subscription>
                <xpath expression="Xpath" persistente="false" replacement="false"/>
                <value>valore</value>
                <xsdref location="Schema Location"/>
            </subscription>
        </wsn-b:TopicExpression>
    </wsn-b:Filter>
</wsn-b:Subscribe>
```

Listing 3.2: `subscribe.xml`

Per quanto riguarda la Publish il messaggio è costituito dal riferimento del client, che tramite un url si identifica al sistema e il contenuto XML che il Publisher vuole effettivamente pubblicare.
Quando il sistema riceve il messaggio estrae il contenuto, il riferimento al Publisher e l'uri dell'XML Schema del contenuto. Qui inizia la seconda fase dell'operazione di Publish; una volta individuato il namespace il sistema dovrà estrarre tutte le Subscription che contengono quel namespace ed una volta estratte provvederà ad eseguire gli Xpath sul contenuto; nel caso in cui l'esecuzione restituisca i valori specificati nella Subscription il sistema inoltrerà il contenuto (e solamente quello) ai Subscriber che hanno visto le loro richieste soddisfatte.
Di seguito un esempio di Publish.

```
<wsn:Notify xmlns:wsn="http://docs.oasis-open.org/wsn/b-2" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance" xsi:schemaLocation="http://docs.oasis-open.org/wsn/br-2.
    xsd" xmlns:food="http://www.ditech.it/tesi/colazione/cibo">
  <wsn:NotificationMessage>
    <wsn:ProducerReference>
      <Address xmlns="http://www.w3.org/2005/08/addressing">
        url del publisher
      </Address>
      <Metadata xmlns="http://www.w3.org/2005/08/addressing">
        <Address>
          Namespace
        </Address>
      </Metadata>
    </wsn:ProducerReference>
    <wsn:Message>
      contenuto
    </wsn:Message>
  </wsn:NotificationMessage>
</wsn:Notify>
```

Listing 3.3: `publish.xml`

Le operazioni Unadvertise e Unsubscription sono speculari alle rispettive Advertise e Subscription e, in quanto tali, riteniamo che non sia rilevante

includere un esempio di listato.

Di seguito un esempio di `ACK` che consente al client di capire come l'operazione è terminata. L'attributo booleano *value* indica l'esito dell'operazione.

```
1 <result
2     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:noNamespaceSchemaLocation="http://ltw0904.web.cs.unibo.it/FlintSchema/Data/
          schema.xsd"
4     valueACK="true"/>
```

Listing 3.4: `Ack.xml`

3.2 Banca Dati

A supporto del sistema in oggetto, è stata sviluppata la banca dati che presentiamo in questa sezione. Analizzando i requisiti, abbiamo individuato le seguenti entità che devono essere modellate e gestite dalla banca dati.

- *Contenuto*: Il contenuto modella un contenuto trasmesso e memorizzato nella base di conoscenza. Esso è istanza di un namespace e può soddisfare zero, una o più espressioni xpath relative a sottoscrizioni;
- *Namespace*: Il namespace modella una famiglia di contenuti;
- *XPath*: Un xpath può modellare una clausola di instradamento oppure una clausola di identificazione. Le clausole di instradamento sono abbinate alle operazioni di tipo subscribe, e si applicano ai contenuti appartenenti ad una determinata famiglia di contenuti per valutare la possibilità di instradamento. Le clausole di identificazione sono invece utilizzate per comprendere se due istanze di una famiglia di contenuti sono di fatto due versioni differenti dello stesso contenuto (nel qual caso la base di conoscenza deve contenere solamente la versione più recente);
- *Cienti*: Un cliente modella un attore che può essere Publisher o Subscriber.

Tali entità si collocano nel modello di dominio illustrato dalla figura 3.1.

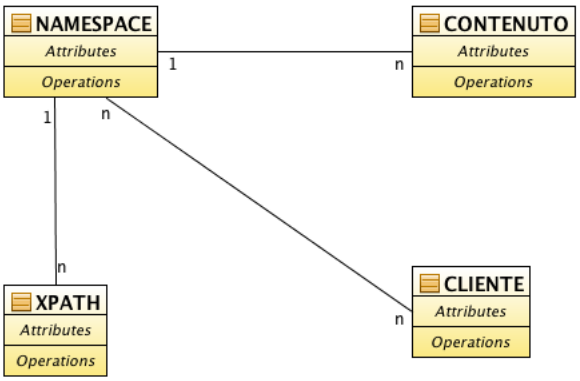

Figura 3.1: *Diagramma delle Entità Fondamentali*

Il modello di dominio evidenzia le seguenti relazioni fondamentali:

- *Cliente-Namespace*: Questa relazione rappresenta il legame che si crea tra un attore, Publisher o Subscriber, e una famiglia di contenuti, a seguito della corretta esecuzione di un'operazione di Advertise o Subscribe, rispettivamente. Un cliente, o attore, può essere legato a zero, uno o più namespace e un namespace può essere legato a zero, uno o più clienti;

- *Namespace-Contenuto*: Questa relazione lega un contenuto alla famiglia a cui esso appartiene. Un contenuto può appartenere ad una e una sola famiglia, ad una famiglia possono appartenere zero, uno o più contenuti;

- *Namespace-Xpath*: Questa relazione collega una famiglia di contenuti con le condizioni che devono essere valutate su tutte le istanze di tale famiglia al fine di valutare le condizioni di instradamento o quelle di replacement. Un namespace può essere in relazione con zero, uno o più

condizioni e una condizione può essere in relazione con uno e un solo namespace;

- *Cliente-Xpath*: Questa relazione raffina la relazione cliente-namespace per consentire ai Subscriber di esprimere condizioni più fini sulle regole di instradamento dei contenuti.

Tutti i concetti elaborati fino ad ora ci hanno permesso di progettare la banca dati che è descritta in figura 3.2.

Ogni messaggio che arriva al sistema, attraversa le tre fasi di logging, storing e forwarding, che presentiamo di seguito. Attraverso la decrizione di queste fasi è possibile comprendere meglio le entità e le relazioni codificate nella banca dati.

3.2.1 Logging

L'attività di logging sfrutta l'omonima tabella per tener traccia di tutte le operazioni invocate sul sistema, associate ad un cliente, un timestamp, e ad un codice che identifica il tipo di operazione. L'operazione implementata in questa fase è comune a tutti i messaggi di protocollo publish/subscribe che implementiamo in questa tesi.

3.2.2 Storing

L'attività di storing, elabora i contenuti di ogni singolo messaggio per popolare coerentemente la banca dati di tutte le informazioni necessarie a implementare le corrette funzionalità di instradamento e mantenimento della base di conoscenza, mantenuta nella tabella content.

Quando il sistema riceve un messaggio di tipo Advertise, si crea nella tabella advertisement una relazione tra un cliente e un namespace, qualificata ad indicare che il cliente è legittimato a inviare al sistema messaggi di publish.

Quando il sistema riceve un messaggio di tipo Subsribe,

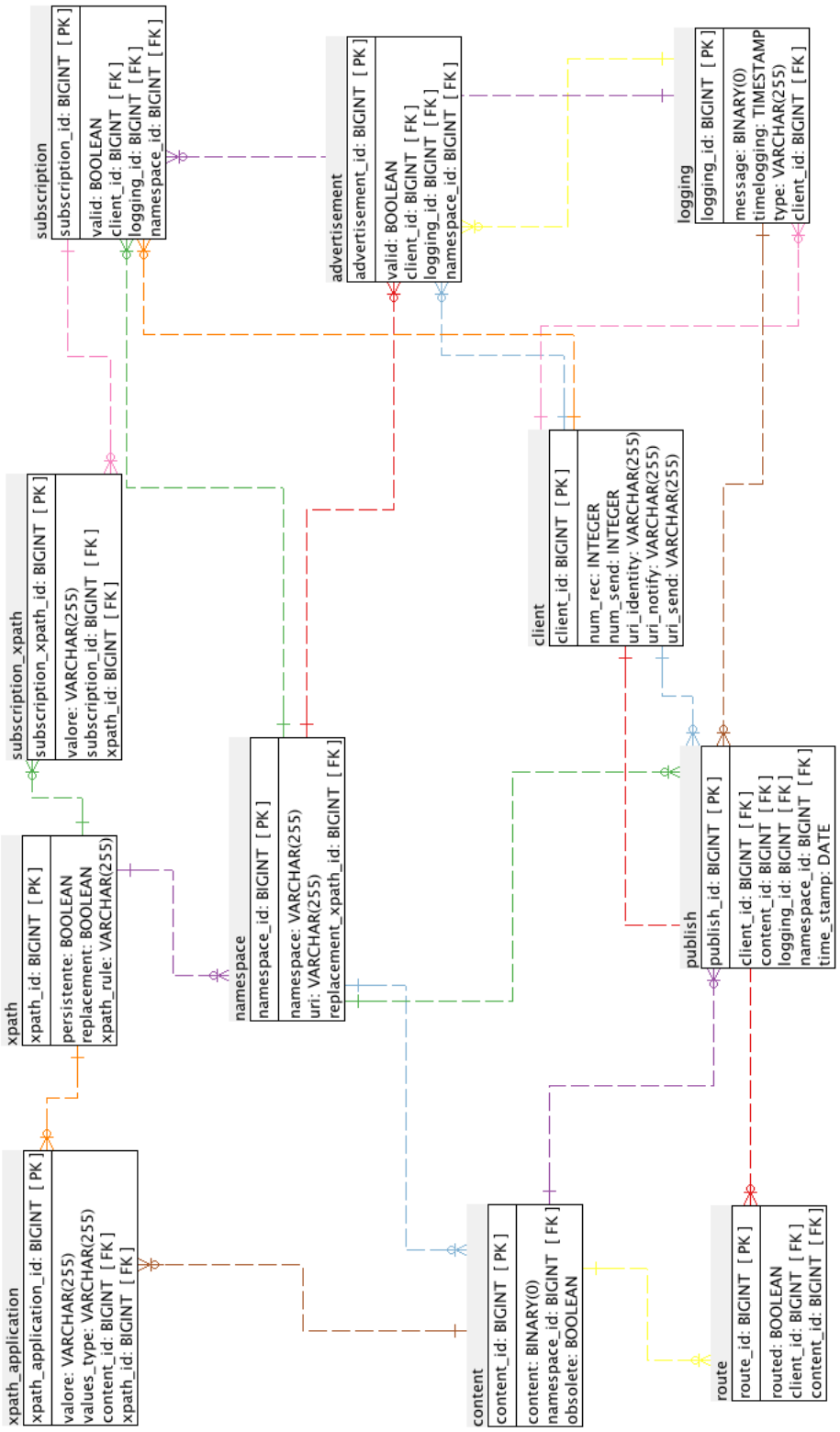

Figura 3.2: *Schema ER Bancadati*

1. si controlla se il cliente non sia già in relazione con lo stesso namespace e con le stesse condizioni, in caso affermativo la subscribe termina restituendo esito positivo;

2. si crea nella tabella subscription una relazione tra un cliente e un namespace, qualificata ad indicare che il cliente è interessato a ricevere contenuti appartenenti alla famiglia identificata dal namespace. Tale relazione sarà legata alle condizioni di instradamento comunicate attraverso il messaggio di tipo subscribe e memorizzate nelle relative tabelle, xpath e values.

Quando il sistema riceve un messaggio di tipo Publish, vengono eseguite le seguenti operazioni:

1. vengono estratti tutti gli xpath associati a criteri di instradamento o di replacement per la famiglia di contenuti;

2. gli xpath individuati vengono applicati al contenuto.

3. Il contenuto viene salvato nella tabella associata e i valori estratti con gli xpath vengono salvati nella tabella `application_xpath` che mette in relazione un contenuto, un xpath definito nel namespace proprio di quel contenuto, e il valore che corrisponde all'applicazione del xpath sul contenuto specifico;

4. i valori associati agli xpath di replacement vengono utilizzati per individuare se nella banca dati esiste un contenuto per cui gli stessi xpath sono associati agli stessi valori. In caso di ritrovamento, tale contenuto è una versione precedente del contenuto appena pubblicato e viene resa obsoleta attraverso l'impostazione del flag `obsolete` associato;

5. dalle tabelle xpath, values, namespace, e subscriptions, vengono individuate tutte le condizioni di instradamento specificate per il namespace che facciano riferimento a sottoscrizioni ancora attive e vengono applicati al contenuto. La query seguente è utilizzata a tal fine.

```
1  select sub.subscription_id, sub.client_id, xml.content_id, count(subx.xpath_id)
2  from xml_contents xml
3  join application_xpath ax
4  on ax.content_id = xml.content_id
5  join subscription sub
6  on sub.namespace_id = xml.namespace_id
7  and sub.valid = 'S'
8  join subscription_xpath subx
9  on subx.subscription_id = sub.subscription_id
10 where ax.xpath_id = subx.xpath_id
11 and ax.valore = subx.valore
12 and xml.content_id = &content_id
13 group by sub.subscription_id, sub.client_id, xml.content_id
14 having count(subx.xpath_id) = (
15      select count(*)
16      from subscription_xpath inner_sub
17      where inner_sub.subscription_id = sub.subscription_id
18 )
```

Listing 3.5: `forwarding.sql`

I risultati individuati dalla query costituiscono le informazioni che saranno scritte nella tabella routing, tabella che mantiene le informazioni di instradamento dei contenuti tra publish e subscriber, in modo da disaccoppiare l'esecuzione dell'operazione di publish dall'esecuzione delle operazioni di notify eventualmente associate. Questa tabella mantiene la relazione tra un contenuto e un cliente a cui, in base alla valutazione delle condizioni di instradamento, si intende notificare il contenuto. Un attributo di questa tabella qualifica lo stato di instradamento. Durante l'operazione di publish, dopo il consolidamento del contenuto nella base di conoscenza, un'interrogazione sulla banca dati viene eseguita per individuare i subscriber autori di tutte le sottoscrizioni soddisfatte nell'applicazione sul contenuto delle condizioni di instradamento. Per ogni subscriber individuato, il contenuto e l'Url del Subscriber vengono inseriti nella tabella di routing e tale record viene marcato con lo stato di instradamento `NOT-ROUTED`, per indicare al servizio di inoltro

che il contenuto nel record deve ancora essere inoltrato al Subscriber, attraverso il URL specificato nella sottoscrizione.

Quando il sistema riceve un messaggio di tipo Unadvertise, la relazione stabilita con la rispettiva advertise, se individuata, viene annullata. In seguito ad un'operazione di tipo unadvertise, ogni operazione di publish sullo stesso namespace da parte dell'attore restituirà un errore.

Analogamente, quando il sistema riceve un messaggio di tipo Unsubsribe, la relazione stabilita con la rispettiva subscribe, se individuata, viene annullata.

3.2.3 Forwarding

Questa fase viene implementata in seguito all'elaborazione di una fase di storing per un'operazione di tipo publish e prende in carico l'operazione di consegna dei contenuti. In questa fase si prende in esame la tabella con le informazioni di instradamento; per ogni contenuto da instradare, vengono estratti i dati necessari e si procede nell'implementazione dell'instradamento. Per ogni contenuto inoltrato, lo stato dell'inoltro viene modificato in base al risultato ottenuto per la notifica del contenuto.

3.3 Componenti e Struttura

In questa sezione spieghiamo come abbiamo progettato un'architettura che implementa le tre fasi di logging, storing, forwarding che abbiamo descritto nella sezione precedente. In particolare abbiamo cercato di progettare l'architettura in maniera che ogni singola fase sia implementata in un componente separato e in modo che ci siano delle interfacce generiche e ben definite per ognuno di tali componenti. In maniera simile, abbiamo cercato di sfruttare tali interfacce per costruirvi un livello di implementazione degli specifici protocolli di trasporto per disaccoppiare l'implementazione della logica di business, realizzata attraverso i componenti presentati in questa

sezione, dall'implementazione dei protocolli di trasporto che possono essere utili per una specifica applicazione del nostro sistema. Questo accorgimento garantisce il soddisfacimento del requisito di *multicanalità*.

La realizzazione di tutti i componenti presentati in seguito si basa sulla piattaforma OpenESB [JS08] e sullo standard JBI [Vin05]. Per ogni compo-

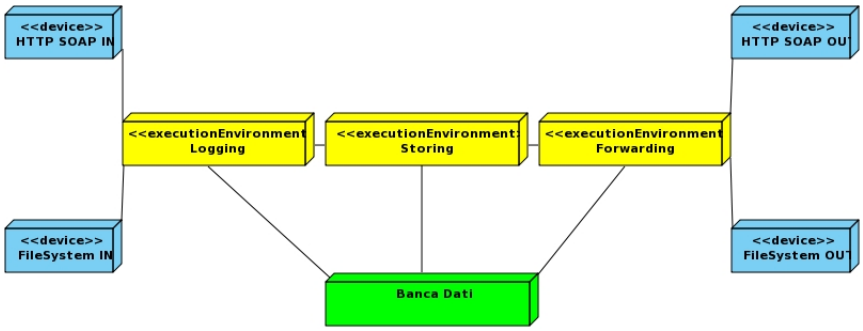

Figura 3.3: *Architettura del sistema*

nente del sistema è stato sviluppato un componente JBI; i componenti che implementano i protocolli di trasporto sono stati realizzati attraverso l'implementazione di Binding Component mentre i componenti che implementano logiche di business sono stati realizzati attraverso l'implementazione di Service Engine.

Per lo scopo di questo prototipo e ai fini del progetto di tesi, abbiamo implementato i Binding Component per gestire lo scambio di messaggi via SOAP/HTTP e via file system. In particolare, per ciascun protocollo di trasporto abbiamo implementato due Binding Component: uno per la gestione dei messaggi in ingresso e l'altro per la gestione dei messaggi in uscita.

OpenESB fa uso di un componente infrastrutturale e configurabile che si occupa di orchestrare l'esecuzione di tutti i servizi. Attraverso un'opportuna configurazione del componente di routing abbiamo implementato la seguente logica di esecuzione del servizio che è comune a tutti i protocolli di trasporto:

1. il binding component inbound riceve un messaggio;

2. il componente di logging viene richiamato;

3. il componente di storing viene richiamato;

4. il componente di forwarding viene richiamato;

5. il binding component outbound provvede alla codifica dell'inoltro sull'opportuno protocollo di trasporto.

Tale approccio progettuale consente di sviluppare ulteriori servizi attraverso ulteriori componenti JBI e combinare tali servizi con quelli esistenti creando logiche di servizio più complesse e garantendo il requisito della *flessibilità*.

E' importante notare che il componente di routing di openESB consente a di configurare i componenti JBI perché possano interagire sia in maniera sincrona che in maniera asincrona. Ai fini del progetto di questo prototipo, avremo comunicazione sincrona tra il binding component inbound e il componente di Logging. Tutte le altre attivazioni saranno effettuate in maniera asincrona, per ottimizzare le performance dell'architettura. Tuttavia, ai fini di avere delle misurazioni che rendano l'idea del livello di ottimizzazione del codice e di performance del servizio, abbiamo deciso ai fini dei test di sfruttare un modello di comunicazione sincrona tra tutti i componenti del sistema, facendo eccezione soltanto per il binding component outbound.

Il componente di Logging riceve il mesaggio da processare e richiama le funzionalità del modulo di mapping del database, schierato come libreria, per salvare il messaggio nell'appropriata tabella di logging relazionandolo col cliente che l'ha inviato. Se il messaggio è corretto e il cliente viene riconosciuto, allora l'operazione termina correttamente restituendo l'identificativo associato al messaggio salvato.

I servizi del componente di Storing vengono richiamati quando i servizi del componente di logging terminano correttamente. Tale servizio prende in esame il messaggio ricevuto, lo suddivide nelle sue componenti e implementa l'operazione attraverso un'opportuna configurazione della banca dati. Come abbiamo visto nella sezione precedente, in questa fase le operazioni svolte sono dipendenti dal messaggio ricevuto.

In merito alla gestione di messaggi di tipo publish, viene richiamato anche il componente di Forwarding; esso ha il compito di analizzare le subscribe effettuate in merito alla famiglia di contenuti a cui quello pubblicato appartiene, analizzare le condizioni di instradamento e istruire il sistema con le informazioni sugli inoltri da effettuare. Se vengono individuati inoltri da effettuare, il componente invia il frammento XML al componente di collegamento che viene ritenuto opportuno in base al subscriber che deve essere notificato. Sarà poi il componente di collegamento ad occuparsi dell'effettivo inoltro.

Un altro componente molto importante è il componente che viene utilizzato per interfacciarsi con la banca dati. Questo componente è strutturato in maniera tale da ricalcare la struttura della banca dati stessa; infatti espone tre metodi per la gestione dei messaggi sul data base e questi metodi consento di eseguire le operazioni di Logging, di Storing e di Forwarding. Il componente viene schierato come una libreria che viene poi richiamata dai tre componenti di logica.

3.4 Robustezza verso attacchi di tipo Denial of Service

In questa sezione presentiamo la tecnica che abbiamo elaborato per contrastare attacchi di tipo Denial of Service (DoS) quando il servizio è esposto attraverso il componente di binding che implementa SOAP/HTTP. Tuttavia, la tecnica individuata è adattabile all'impiego con altri protocolli di trasporto.

A livello applicativo, attacchi di tipo DoS si possono prevenire impedendo che le richieste applicative provochino esaurimento della memoria a disposizione del servizio. La memoria a disposizione di un'applicazione può esaurirsi quando una richiesta operativa richiede più memoria di quella a disposizione oppure quando la totalità di richieste attualmente servite esauriscono la memoria disponibile.

Per prevenire entrambe le tipologie di attacco, abbiamo adottato i seguenti accorgimenti:

- *Una richiesta esaurisce la memoria*: Le operazioni di servizi web che ammettono parametri di dimensione arbitraria si prestano molto ad attacchi di tipo DoS; infatti, i parametri delle invocazioni alle operazioni vengono collocati direttamente sullo heap della jvm ed un parametro di dimensione sufficientemente elevata è in grado di esaurire la memoria a disposizione della macchina virtuale. Per individuare questo tipo di possibilità prima che il servizio subisca un guasto, abbiamo deciso di veicolare i contenuti attraverso attachment mime abbinati ai messaggi SOAP. In questo modo i contenuti non vengono caricati direttamente in memoria principale ma vengono mantenuti nel buffer dello stream HTTP e possono essere recuperati esplicitamente da programma quando è necessario elaborarli e dopo eventuali controlli. Questa tecnica di salvataggio e recupero dei contenuti previene l'esaurimento di memoria in ricezione di un messaggio operativo. A seguito di questa modifica, il binding component inbound si occupa di memorizzare l'attachment in un file e di passare il riferimento del file al logging component; sarà responsabilità di quest'ultimo individuare contenuti illegittimi e scartarli prima di iniziarne l'elaborazione (di questo aspetto ne parliamo al punto successivo).

- *Le richieste in elaborazione esauriscono la memoria*: Per evitare questa tipologia di errore abbiamo individuato la necessità di razionare le risorse che vengono utilizzate per ogni messaggio operativo servito. Questa tecnica è realizzabile in quanto è ragionevole assumere che un contenuto, data la sua natura di record di metadati, possa avere una dimensione massima e che quindi sia limitata la memoria richiesta dalla sua elaborazione. Dobbiamo quindi riuscire a imporre che la memoria utilizzata a runtime dal server sia limitata; per fare questo possiamo avvalerci di due tecniche:

- *Limitazione della memoria usata per operazione*: A tal fine abbiamo imposto un controllo nella fase di logging prima del termine della logica implementata dal componente associato, che previene all'operazione di publish di essere elaborata se il contenuto ha una dimensione che eccede il massimo consentito. In questo caso l'attività di questa fase termina restituendo un codice di insuccesso. La dimensione massima del contenuto è configurabile nel file contenente le impostazioni del servizio;
- *Tuning dell'application server*: Gli application server della famiglia J2EE hanno un grado di configurabilità tale che è possibile imporre una limitazione al numero di worker thread che servono le richieste di una determinata applicazione. Questo consente in fase di deploy di sapere quanti contenuti possono essere elaborati concorrentemente;

Possiamo quindi approssimare la memoria massima usata dal server a runtime come la somma tra la memoria necessaria alle singole parti del server per funzionare più il prodotto tra il numero massimo di thread e il risultato della somma tra la dimensione massima del contenuto e una costante che rappresenta la memoria aggiuntiva necessaria a portare a termine l'elaborazione di tale contenuto. Attraverso una scelta oculata dei valori dei parametri che rappresentano la dimensione massima del contenuto e il numero di worker thread per il nostro servizio, siamo in grado di garantire la prevenzione degli attacchi DoS che esauriscono la memoria del servizio.

Eventuali altri attacchi perpetrati con strumenti più a basso livello saranno da risolvere attraverso opportuni stratagemmi di natura infrastrutturale o sistemistica e, pertanto, esulano dalla portata di questa tesi.

Capitolo 4

Sviluppo

Introduzione

In questo capitolo mostreremo gli aspetti fondamentali dello sviluppo del progetto.

Il primo aspetto trattato sarà il protocollo di comunicazione che abbiamo utilizzato per la trasmissione dei dati e in particolare ci soffermeremo sulla mappatura della semantica.

Le varie operazioni definite in precedenza saranno codificate con il protocollo XML scelto.

Definito questo aspetto prenderemo in esame la parte infrastrutturale e di collegamento per la gestione della comunicazione e delle logiche del sistema.

Questa parte di infrasrtuttura si basa su una piattaforma scelta ad hoc atta a creare un ambiente modulare e orientato ai servizi.

In ultimo la parte che verrà trattata riguarda la logica di sistema e la gestione della banca dati.

4.1 Protocollo di Comunicazione XML

Come già spiegato nel capitolo precedente abbiamo utilizzato il protocollo WS-BaseNotification [SGM06].

Di seguito mostriamo la mappatura diretta dell'interfaccia del sistema, tramite il sopracitato protocollo XML.

Operazione	Messaggio	Namespace
Advertise	RegisterPublisher	`http://docs.oasis-open.org/wsn/br-2`
Subscribe	SubscribeRequest	`http://docs.oasis-open.org/wsn/b-2`
Publish	Notify	`http://docs.oasis-open.org/wsn/b-2`
Unsubscribe	Unsubscribe	`http://docs.oasis-open.org/wsn/b-2`
Unadvertise	DestroyRegistration	`http://docs.oasis-open.org/wsn/br-2`

Tabella 4.1: *Mappatura delle operazioni di sistema*

Come possiamo notare la mappatura dell'interfaccia è diretta per ogni operazione ad eccezione della nomemenclatura.

Questo ci ha permesso di implementare un'interfaccia robusta basata su uno standard largamente impiegato.

Per quanto riguarda la gestione e la manipolazione dei file WSDL ne parleremo ampiamente quando tratteremo la piattaforma e l'architettura del sistema.

4.2 Piattaforma di Sviluppo e Componenti

Come piattaforma di sviluppo abbiamo scelto GlassFishESB di Sun Microsystem, che integra l'Application Server GlassFish e l'esb OpenESB.

GlassFishESB è scritto in Java ed aderente alle specifiche Java EE 6.

La funzionalità principale dell'ESB [Vin05] è quella di mettere a disposizione dello sviluppatore una piattaforma di integrazione SOA che consenta di sviluppare dei moduli indipendenti adibiti , ognuno, ad un servizio indipendente.

I moduli saranno poi orchestrati, a formare un unico servizio complesso. Le tipologie dei moduli sono quattro:

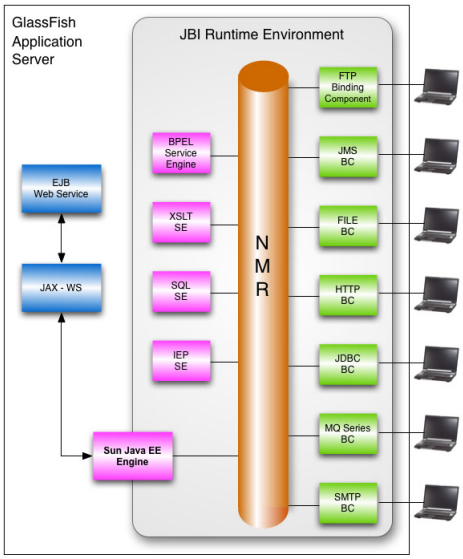

Figura 4.1: *Architettura Open ESB*

1. **Service Engine**: moduli pluggabili nella piattaforma, implementano dei servizi che possono poi essere orchestrati insieme.

2. **Binding Component**: moduli simili ai Service Engine ma svolgono solamente funzioni di collegamento con l'esterno. Ogni Binding Component gestisce un protocollo di comunicazione.

3. **Shared Library**: moduli di libreria condivisi da più Service Engine.

4. **Service Asseblies**: moduli di configurazione che servono a collegare più Service Engine e Binding Component a creare un unico servizio complesso. Sono i responsabili dell'orchestrazione.

Nella realizzazione del sistema abbiamo realizzato, attraverso un service assemblies, un servizio multicanale impiegando tre Service Engine, due Binding Component e un modulo di libreria condivisa.

La struttura si fonda sullo standard JBI che definisce un'architettura per uniformare l'integrazione di componenti EIS eterogenei all'interno di applicazioni JBI- compliant.

Le specifiche JBI definiscono un ambiente Java per l'integrazione basato sui principali standard open e di mercato. JBI definisce uno standard mirato ai container che permette lo sviluppo di container di servizi secondo un modello a plugin. Lo scenario che si prospetta è la definizione di un contenitore (l'environment JBI) in grado di ospitare "container di servizi" agendo da "container of containers" in grado di interagire mediante un sistema di messaging basato sul Web Services Description Language 2.0.

L'idea di base è di poter definire dei contenitori di servizi che consentano la System Integration e permettere quindi a sistemi inizialmente non progettati per lavorare insieme di cooperare tra loro come se fossero un'unica applicazione (Composite Application).

In questo modo lo sviluppatore sarà in grado di creare un'applicazione "assemblando" le funzionalità necessarie utilizzando gli opportuni Componenti plugin JBI.

4.2.1 Service Engine

I Service Engine (SE) sono componenti JBI che forniscono logica di integrazione e di trasformazione verso altri componenti così come a loro volta possono utilizzare i servizi di altri SE. I SE sono in grado principalmente di integrare applicazioni/risorse Java-Based.

La struttura del pacchetto .jar che verrà installato sul Service Bus è molto semplice e consiste nei package Java contenenti i file .class e una directory META-INF che contiene il file di configurazione jbi.xml.

La funzione del file di configurazione è quella di indicare le classi che implementano le interfacce obbligatorie del container JBI.

Le interfacce sono tre e si occupano di gestire le fasi del ciclo di vita del componente: installazione, rimozione, avvio e spegnimento. Nel file jbi.xml vengono anche specificate le librerie condivise che il componente utilizza.

Più nel dettaglio la parte dei file .class va impacchettata in un .jar il cui nome deve essere specificato nel file jbi.xml.

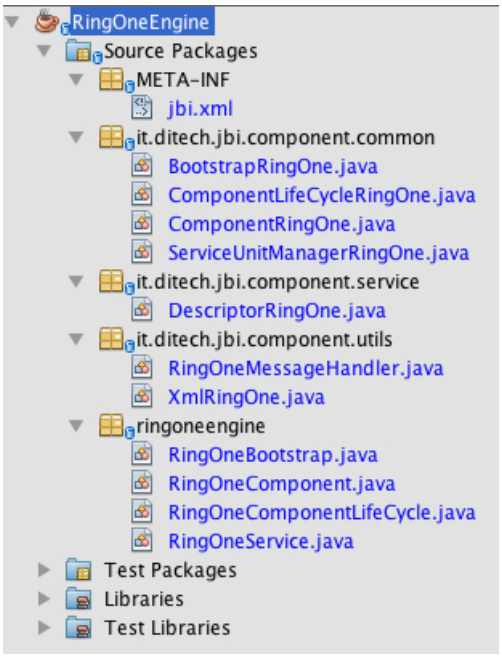

Figura 4.2: *Struttura del componente*

La figura 4.2 riguarda la struttura del primo Service Engine, che svolge la funzione di logging dei messaggi.

Gli altri due Service Engine rispettivamente RingTwoEngine e RingThreeEngine svolgono le funzioni di storing ed inoltro dei messaggi; ma la struttura è la medesima. Vediamo ora i due package fondamentali:

`it.ditech.jbi.component` : questo package contiene altri tre package nell'ordine: `common`, `service` e `utils`.

Il primo contiene le quattro classi fondamentali che implementano le interfacce del container JBI che sono `Bootstrap` (implemetata nella classe BootstrapRingOne.java), per la gestione del boot del componente, `ComponentLifeCycle` (implemetata nella classe ComponentLifeCycleRingOne.java) che serve a gestire l'intero ciclo di vita del componente, `Component` (implemetata nella classe ComponentRingOne.java) che serve a stabilire la struttura fondamentale ed infine l'interfaccia `ServiceUnitManager` (implemetata nella classe ServiceUnitManager-

RingOne.java) che svolge la funzione di definizione del servizio che il componente espone.

Il secondo packege contenuto in `component` è `service`, in questo package si trova una classe che implementa il descrittore del servizio dell'engine.

Il package `utils` fornisce delle utiliy per la gestione dell'XML e della fase di scambio di messaggi tra l'engine in questione e gli altri componenti.

`ringoneengine` : questo è il package che estende le classi del package `it.ditech.jbi.component.common` e che devono essere riportate nel file jbi.xml. Le classi RingOneBootstrap.java, RingOneComponent.java e RingOneComponentLifeCycle.java estendono rispettivamente le classi BootstrapRingOne.java, ComponentRingOne.java e ComponentLifeCycleRingOne.java.

La classe ServiceRingOne.java è la classe in cui viene implementato il servizio vero e proprio ed attraverso i metodi della classe vengono richiamati i servizi. Per separare infrastruttura e logica abbiamo deciso di spostare quest'ultima all'interno di una libreria condivisa che verrà richiamata dall'interno di ogni engine.

Di seguito riportiamo un esempio del file jbi.xml, inerente al primo Service Engine.

```
1  <jbi version="1.0" xmlns="http://java.sun.com/xml/ns/jbi">
2      <component type="service-engine">
3          <identification>
4              <name>RingOneEngine</name>
5              <description>Primo Engine della catena</description>
6          </identification>
7          <component-class-name>ringoneengine.RingOneComponent</component-class-name>
8          <component-class-path>
9              <path-element>componente.jar</path-element>
10         </component-class-path>
```

```
11          <bootstrap-class-name>ringoneengine.RingOneBootstrap</bootstrap-class-name>
12          <bootstrap-class-path>
13              <path-element>componente.jar</path-element>
14          </bootstrap-class-path>
15          <shared-library>ditech-ejb-libraries</shared-library>
16      </component>
17  </jbi>
```

Listing 4.1: `jbi.xml`

Come si può notare il file riporta le informazioni necessarie per il deploy e l'installazione del componente, in particolare indica dove si trova la classe che implementa l'interfaccia javax.jbi.component.Bootstrap e l'interfaccia javax.jbi.component.ComponentLifeCycle.

Le altre informazioni riguardano il nome del componente e le librerie utilizzate da quest'ultimo.

L'ultimo aspetto che andremo a trattare per i Service Engine riguarda lo scambio di messaggi con altri componenti. Lo scambio dei messaggi viene definito nello standard JBI e vengono implementati diversi pattern di scambio; nel nostro caso è stato usato il pattern *InOut*, che implementa una comunicazione bloccante sincrona. Di seguito il codice Java del metodo utilizzato nella classe `RingOneMessageHandler` per l'inoltro dei messaggi:

```
1  public static String doInOutMessageExchange(Long id, RingOneComponentLifeCycle compRing) throws
       Exception {
2
3          String result = "";
4          channel = compRing.getDeliveryChannel();
5          ServiceEndpoint serviceEndpoint = findServiceEndpoint(compRing);
6          if (serviceEndpoint == null) {
7              compRing.getLogger().info("[doInOutMessageExchange]: Il primo Service Engine non ha
                   trovato nessun ServiceEndpoint attivo per");
8          }
9          QName operation = new QName("http://www.ditech.it/jbi/tesi/santi/roberto", "stepTwoOperation"
               );
10         InOut inOutMe = createInOutMessageExchange(operation, serviceEndpoint, compRing, channel);
11         //Setta un NM sul canale
12         NormalizedMessage inMsg = inOutMe.createMessage();
13         //Setta il contenuto del messaggio
14         inMsg.setContent(null);
15         String identificatore = String.valueOf(id);
16         inMsg.addAttachment("allegato", new DataHandler(identificatore, "text/plain"));
17         //invia il messaggio al Service Engine Successivo
18         inOutMe.setInMessage(inMsg);
19         if (channel.sendSync(inOutMe, RingOneComponentLifeCycle.SEND_SYNC_TIMEOUT)) {
20             Logger.getLogger("RingOneMessageHandler").log(Level.INFO, "INOLTRO ESEGUITO CORRETTAMENTE
                   ");
21         }
```

```
22          ExchangeStatus status = inOutMe.getStatus();
23          if (ExchangeStatus.ERROR.equals(status)) {
24              throw new Exception("[doInOutMessageExchange]: Errore di comunicazione tra il primo e il
                    secondo Service Engine");
25          }
26          //Recupera la risposta
27          NormalizedMessage outMsg = inOutMe.getOutMessage();
28          DataHandler dh = outMsg.getAttachment("result");
29          result = (String) dh.getContent();
30          dh = null;
31
32          if (inOutMe.getStatus().equals(ExchangeStatus.ERROR)) {
33              inOutMe = null;
34              inMsg = null;
35              outMsg = null;
36              result = "ERROR";
37
38          } else {
39              inOutMe.setStatus(ExchangeStatus.DONE);
40              inOutMe = null;
41              inMsg = null;
42              outMsg = null;
43          }
44          return result;
45  }
```

Listing 4.2: `message.java`

Le righe di codice più significative sono la riga 10, la riga 19, la riga 27 e la riga 39. Alla riga si 10 utilizza il metodo `createInOutMessageExchange` per ottenere un oggetto `MessageExchange` il responsabile dello scambio dei messaggi. Tale oggetto viene creato a partire dal nome dell'operazione da invocare sull'altro componente, dal canale aperto verso il componente destinazione del messaggio e dal `ServiceEndpoint` ovvero l'endpoint del Service Engine ricevente.

Alla riga 19 sull'oggetto `MessageExchange` viene richiamato il metodo `sendSync` che inoltra il messaggio[1] e restituisce un `boolean` in base all'esito dell'inoltro.

Se l'esito è `false` il metodo termina con errore, altrimenti continua dalla riga 27. Con il metodo `getOutMessage()` viene recuperata la risposta del componente remoto. In ultimo alla riga 39 viene settato lo stato del `MessageExchange` a `DONE` ovvero lo scambio è terminato correttamente. A questo punto il metodo termina con esito positivo.

Per fare una maggiore chiarezza possiamo vedere di seguito una rappresentazione dello scambio di messaggi basato sul pattern InOut.

[1] *Il metodo inoltra il messaggio in modalità sincrona e utilizza un intervallo in ms per la sincronizzazione. Se entro quell'intervallo non è giunta nessuna risposta viene sollevata un'eccezione di time out e viene retituito un false*

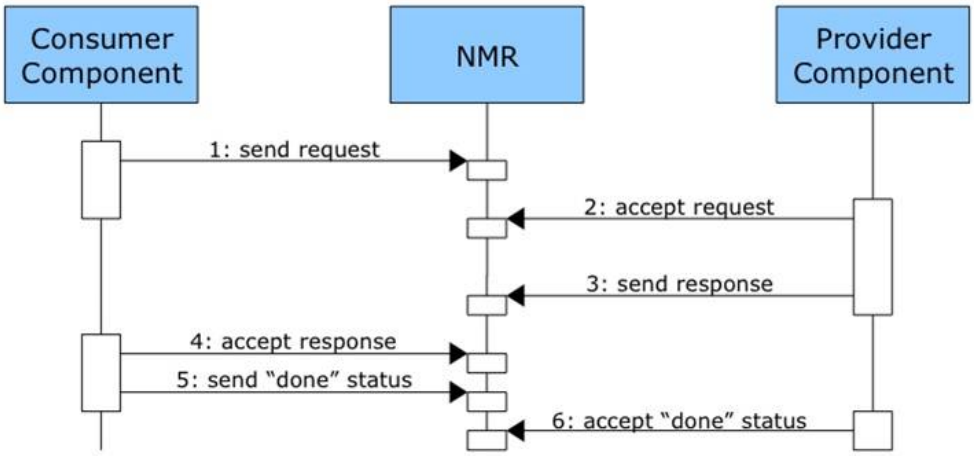

Figura 4.3: *Message Exchange Pattern InOut*

Ogni frammento XML ricevuto da un qualsiasi Binding Component viene adattato al formato *Normalized Message* ed inoltrato ad altri componenti, non direttamente bensì attraverso un componente infrastrutturale del container JBI il *Message Normalized Router* (Figura 4.4).

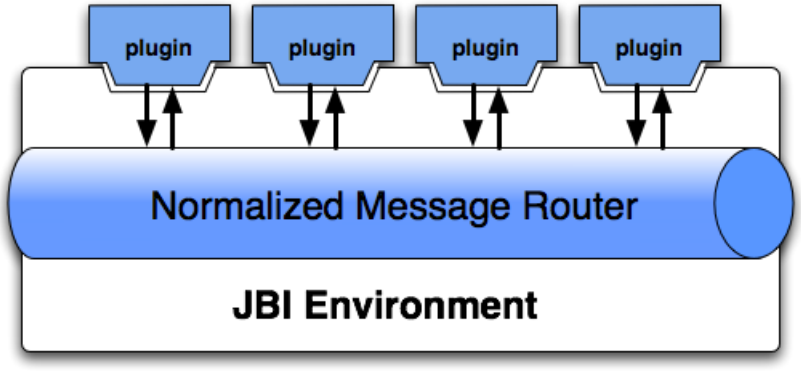

Figura 4.4: *JBI Message Normalized Router*

4.2.2 Binding Component

I Binding Components (BC) hanno lo scopo di fornire la logica di connettività da/verso i servizi esterni all'installazione JBI. Il compito principale dei BC è quindi di integrare applicazioni/Enterprise Information Systems (EIS) non Java-based. Il compito principale dei BC è quindi quello di normalizzare e/o denormalizzare i messaggi da/verso i servizi esterni adattando i protocolli/formati deli servizi remoti esterni all'ambiente JBI (es: HTTP, SOAP, JMS, JCA, FTP, TCP/IP, AS1/AS2-EDI,...).

In questo modo è possibile per i servizi Service Engine (SE) interni all'environment JBI sia consumare i servizi remoti esterni a JBI (comunicazione inbound) che esportare le proprie funzionalità ai servizi remoti (comunicazioni outbound).

Ricapitolando: i BC devono occuparsi di adattare i messaggi in entrambi le direzioni:

nelle comunicazioni inbound devono normalizzare i messaggi trasformando il formato specifico del protocollo di comunicazione e dei dati in messaggi in forma canonica.

nelle comunicazione outbound devono invece denormalizzare il messaggio trasformando il messaggio normalizzato nel formato specifico dei dati del client esterno all'ambiente JBI.

Nell'immagine che segue possiamo notare come i Binding Component lavorino a livello dei messaggi nel container JBI.

Nel nostro progetto sono stati utilizzati due tipi di BC in entrambi i casi sia l'inbound che per l'outbound.

Il primo BC è per il protocollo HTTP/SOAP ovvero per ricevere ed inoltrare messaggi SOAP. La configurazione del BC viene fatta attraverso dei file wsdl che andranno inpacchettati e deploiati nel service assembly.

L'altro BC invece è usato per leggere e scrivere file da una cartella del FileSystem. Come il precedente, viene configurato attraverso dei file wsdl da inserire nel deploy del service assembly.

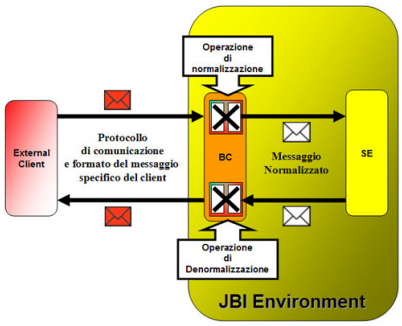

Figura 4.5: *Funzionamento di un Binding Component*

A differenza dei SE i Binding Component sono già sviluppati ed integrati nella piattaforma e quindi necessitano solamente di essere configurati senza bisogno di scrivere del codice Java.

4.2.3 Shared Library

Le Shared Library sono delle librerie condivise messe a disposizione come API all'interno del container JBI. Il modulo contenete la logica di business del nostro sistema è totalmente sviluppato e schierato in una libreria condivisa e i SE utilizzano i metodi di logica richiamando le API di questa libreria.

Il deploy di una Shared Library consiste in un file .zip composto da un .jar, che contiene il modulo di libreria vero e proprio, e da una cartella META- INF che contiene il descrittore di deploy jbi.xml.

Di seguito il listato del file jbi.xml per il deploy della nostra Shared Library:

```
1  <jbi
2      xmlns="http://java.sun.com/xml/ns/jbi" version="1.0"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xmlns:identification="http://www.sun.com/jbi/descriptor/identification/v1.0">
5
6      <shared-library class_loader_delegation="self-first">
7          <identification>
8              <name>ditech-ejb-libraries</name>
9              <description>ditech-ejb-libraries</description>
10             <identification:VersionInfo component_version="1.0.0" build-number="000001"
                   />
11         </identification>
12         <shared-library-class-path>
13             <path-element>PubSubEJB.jar</path-element>
14         </shared-library-class-path>
15     </shared-library>
16 </jbi>
```

Listing 4.3: `jbi.xml`

Il file è diviso in due parti fondamentali l'identificazione e il classpath della libreria.

L'identificazione consiste nel nome della libreria ed alcune informazioni sulla versione del componente; nella parte path invece vengono specificati i classpath degli eventuali .jar di libreria. Ovviamente il numero dei file .jar è illimitato.

PubSubEJB.jar è il modulo nel quale è stata sviluppata l'intera logica del sistema.

4.2.4 Service Assembly

Un Service Assembly è un pacchetto di configurazione che serve a comporre una serie di moduli a formare un servizio finale articolato e complesso. Questo pacchetto ha una struttura interna gerarchica ed è organizzato come mostrato in Figura 4.6

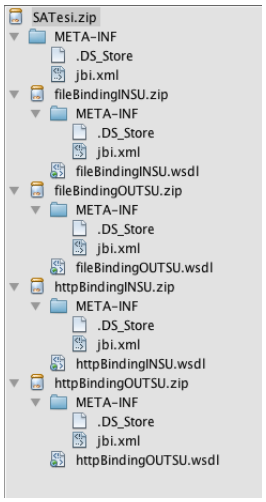

Figura 4.6: *Struttura del Service Assembly*

L'immagine si riferisce al Service Assembly SATesi sviluppato per il prototipo di tesi.

Il primo livello è composto da quattro file .zip che contengono i descrittori di servizio dei Binding Component utilizzati nel progetto mentre la cartella META-INF contiene il file jbi.xml che descrive l'intera struttura del service assembly.

Di seguito mostriamo il descrittore del service assembly jbi.xml.

```
1 <jbi version="1.0"
2     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xmlns="http://java.sun.com/xml/ns/jbi"
4     xmlns:wsn-brw="http://docs.oasis-open.org/wsn/brw-2">
5     <service-assembly>
6         <identification>
7             <name>JBIServiceTesi</name>
8             <description>Service Assembly del prototipo di tesi</description>
9         </identification>
```

```
10          <service-unit>
11              <identification>
12                  <name>httpBindingINSU</name>
13                  <description>Binding Component IN SOAP/HTTP</description>
14              </identification>
15              <target>
16                  <artifacts-zip>httpBindingINSU</artifacts-zip>
17                  <component-name>sun_http_binding</component-name>
18              </target>
19          </service-unit>
20          <service-unit>
21              <identification>
22                  <name>httpBindingOUTSU</name>
23                  <description>Binding Component OUT SOAP/HTTP</description>
24              </identification>
25              <target>
26                  <artifacts-zip>httpBindingOUTSU</artifacts-zip>
27                  <component-name>sun_http_binding</component-name>
28              </target>
29          </service-unit>
30          <service-unit>
31              <identification>
32                  <name>fileBindingOUTSU</name>
33                  <description>Binding Component OUT FILE</description>
34              </identification>
35              <target>
36                  <artifacts-zip>fileBindingOUTSU</artifacts-zip>
37                  <component-name>sun_file_binding</component-name>
38              </target>
39          </service-unit>
40          <service-unit>
41              <identification>
42                  <name>fileBindingINSU</name>
43                  <description>Binding Component IN FILE</description>
44              </identification>
45              <target>
46                  <artifacts-zip>fileBindingINSU</artifacts-zip>
47                  <component-name>sun_file_binding</component-name>
48              </target>
49          </service-unit>
50          <connections>
51              <connection>
52                  <consumer endpoint-name="portFile" service-name="wsn-brw:
```

```
                    echoServiceFile"/>
53                <provider endpoint-name="echoEP_JBIPort" service-name="wsn-brw:
                    echoService"/>
54            </connection>
55            <connection>
56                <consumer endpoint-name="port" service-name="wsn-brw:echoService"/>
57                <provider endpoint-name="echoEP_JBIPort" service-name="wsn-brw:
                    echoService"/>
58            </connection>
59        </connections>
60    </service-assembly>
61 </jbi>
```

Listing 4.4: `jbi.xml`

Come si può vedere il file individua quattro service unit che costituisco un'unità singola di collegamento.

Due di queste unità utilizzano il componente `sun-http-binding` mentre le altre due utilizzano il componente `sun-file-binding`. In ogni coppia esiste un modulo di IN e un modulo di OUT.

La parte finale del file invece indica quali sono gli endpoint del servizio. Ne esistono due con il ruolo di consumer, che sono i responsabili dell'inoltro messaggi verso l'interno (`port` e `portFile`), mentre entrambe le SU di Input inoltrano i messaggi normalizzati allo stesso endpoint interno al container JBI: `echoEP-JBIPort` che ovviamente ha il ruolo di provider.

Gli endpoint esterni delle SU, con funzione di provider, vengono definiti all'interno dei file WSDL mostrati nel listato 4.5.

La parte interssante dei file WSDL è la parte che riguarda il servizio e il binding, infatti trattandosi di componenti che non lavorano solo con il protocollo HTTP/S diventa interessante guardare la struttura e la forma del binding. Per chiarezza di seguito mostriamo il frammento WSDL del SU `fileBindingINSU` per quanto riguarda il binding ed il servizio.

```
1     ...
2 <binding name="FileBinding" type="wsn-brw:NotificationBroker">
3   <file:binding/>
4   <operation name="Notify">
5     <file:operation verb="poll"/>
6     <wsdl:input name="Notify">
```

```
7       <file:message fileName="input.xml" pollingInterval="1000" lockName="filebc1.lck"
            use="literal" part="Notify"/>
8     </wsdl:input>
9   </operation>
10  <operation name="Subscribe">
11    <file:operation verb="poll"/>
12    <wsdl:input name="Subscribe">
13      <file:message fileName="input.xml" pollingInterval="1000" lockName="filebc2.lck"
            use="literal" part="SubscribeRequest"/>
14    </wsdl:input>
15  </operation>
16  <operation name="GetCurrentMessage">
17    <file:operation verb="poll"/>
18    <wsdl:input name="GetCurrentMessage">
19      <file:message fileName="input.xml" pollingInterval="1000" lockName="filebc3.lck"
            use="literal" part="GetCurrentMessageRequest"/>
20    </wsdl:input>
21  </operation>
22  <operation name="RegisterPublisher">
23    <file:operation verb="poll"/>
24    <wsdl:input name="RegisterPublisher">
25      <file:message fileName="input.xml" pollingInterval="1000" lockName="filebc4.lck"
            use="literal" part="RegisterPublisherRequest"/>
26    </wsdl:input>
27  </operation>
28 </binding>
29 <service name="echoServiceFile">
30   <port name="portFile" binding="wsn-brw:FileBinding">
31     <file:address fileDirectory="/usr/local/santi/out-in"/>
32   </port>
33 </service>
34     ...
```

Listing 4.5: `fileBindingINSU.wsdl`

La specifica dell'operazione utilizza l'estensione wsdl `file` e dichiara con l'attributo `verb` il tipo di operazione, ovvero `poll`. Questo attributo sta ad indicare che il BC dovrà eseguire un polling, su una data directory specificata nella parte di definizione del servizio.

Le caratteristiche del polling indicano di eseguire un'operazione di poll ogni 1000 ms e di prelevare un file di nome input.xml andando a ricercare un elemento

XML con il nome specificato dall'attributo `part`.

L'attributo `lockName` invece specifica il nome del file di lock sulla directory che viene creato ogni qualvolta il BC tenta di leggere su quella cartella. Questo evita il sovrapporsi di letture sulla cartella.

Con l'elemento `address` e l'attributo `fileDirectory` viene indicato il path assoluto della cartella sulla quale il BC esegue il polling; come indicato nella definizione del servizio.

In maniera del tutto analoga avviene la definizione delle caratteristiche dell'unità di binding di Out per il BC `sun-file-binding`.

L'unico aspetto interessante, riguardante il WSDL per la configurazione del servizio SOAP/HTTP, riguarda l'utilizzo degli attachment MIME.

Infatti come si è detto parlando di DoS i messaggi SOAP non trasportano i contenuti direttamente nel body, ma utilizzano gli attachment mime.

Essendo il WSDL del tutto analogo a quello di un semplice WebService SOAP, che utilizza gli attachment, ci riserviamo di non mostrare il listato del file.

4.3 Logica e Persistenza

In questa sezione analizzeremo la struttura e le funzionalità di logica di business.

Come detto in precedenza tutte le logiche del sistema sono state riunite in un unico modulo e messo a disposizione dei SE come libreria condivisa.

Il modulo chiamato PubSubEJB implementa la gestione completa della banca dati. Per approcciare a questa problematica in maniera strutturata e flessibile abbiamo scelto di utilizzare un ORM che ci consenta di mappare l'intera banca dati (relazioni e tabelle) con oggetti Java mantenendo quindi un'omogeneità nella gestine gestione delle risorse.

Come ORM abbiamo scelto di utilizzare Hibernate essendo il più diffuso e il più evoluto tra i vari ORM disponibili per il linguaggio Java.

Per la mappatura vera e propria e la persistenza abbiamo utilizzato le API JPA e come DBMS abbiamo adottato PostgreSQL nella versione 8.4.

4.3.1 Tabelle Relazioni ed Entity Bean

Con la progettazione della base di conoscenza abbiamo ottenuto un set di tabelle che consente di avere un mappaggio completo di tutte le informazioni che il sistema ha necessità di gestire e di utilizzare. L'elenco di tabelle che segue è stato generato a partire dallo schema ER ottenuto nella fase di progettazione e, in base a questo, sono stati generati tutti gli entiy bean. La sequence `hibernate_sequence` è stata generata al momento del deploy per la generazione delle chiavi primarie da parte del provider di Hibernate.

Schema	Name	Type	Owner
public	advertisement	table	jbi_user
public	xpath_application	table	jbi_user
public	client	table	jbi_user
public	content	table	jbi_user
public	hibernate_sequence	sequence	jbi_user
public	logging	table	jbi_user
public	namespace	table	jbi_user
public	publish	table	jbi_user
public	route	table	jbi_user
public	subscription	table	jbi_user
public	subscription_xpath	table	jbi_user
public	xpath	table	jbi_user

Tabella 4.2: *Elenco delle tabelle della banca dati*

Come si può vedere dalla tabella tutte le relazioni hanno lo stesso proprietario, jbi_user, che è l'utente che l'ORM utilizza per connettersi al DBMS.

Utilizzando le funzionalità di NetBeansIDE 6.8 ci siamo generati in maniera automatica tutti gli Entity Bean. Inoltre si è scelto di utilizzare l'oggetto Java List<...> per mappare le relazioni tra tabelle con cardinalità superiore ad 1. Di seguito mostriamo un entity bean generato da Hibernate ed andremo ad analizzarne il contenuto.

```
1 package it.ditech.ejb.entity;
```

```
2
3       ...
4
5  /**
6   *
7   * @author Roberto Santi
8   */
9  @Entity
10 @Table(name = "namespace", catalog = "jbi_database", schema = "public")
11 @NamedQueries({@NamedQuery(name = "Namespace.findAll", query = "SELECT n FROM Namespace n")})
12 public class Namespace implements Serializable {
13     private static final long serialVersionUID = 1L;
14     @Id
15     @GeneratedValue(strategy=GenerationType.SEQUENCE)
16     @Basic(optional = false)
17     @Column(name = "namespace_id", nullable = false)
18     private Long namespaceId;
19     @Basic(optional = false)
20     @Column(name = "namespace", nullable = false, length = 255)
21     private String namespace;
22     @Basic(optional = false)
23     @Column(name = "uri", nullable = false, length = 255)
24     private String uri;
25     @OneToMany(cascade = CascadeType.ALL, mappedBy = "namespaceId")
26     private List<Subscription> subscriptionList;
27     @OneToMany(cascade = CascadeType.ALL, mappedBy = "namespaceId")
28     private List<Advertisement> advertisementList;
29     @OneToMany(cascade = CascadeType.ALL, mappedBy = "namespaceId")
30     private List<Publish> publishList;
31     @JoinColumn(name = "replacement_xpath_id", referencedColumnName = "xpath_id", nullable = false)
32     @ManyToOne(optional = false)
33     private Xpath replacementXpathId;
34
35     ...
```

Listing 4.6: `Namespace.java`

Come si può vedere sono state utilizzate le annotazioni JPA per la mappatura delle tabelle e per la creazione dei vari campi del bean.

Per quanto riguarda le relazioni uno-a-molti è stato utilizzato l'oggetto List<...>, che consente di accedere in maniera sequenziale e random alle entità riferite, attraverso il casting di oggetti Java.

Per le realazioni molti-a-uno si è utilizzata l'annotazione *@JoinColumn* in modo tale da avere un riferimento per le colonne da utilizzare per referenziare una data tabella.
In ultimo mostriamo il file di configurazione della persistenza usato per mappare le entità e configurare il comportamento di Hibernate nella gestione della banca dati.

```
1 <persistence version="1.0"
2   xmlns="http://java.sun.com/xml/ns/persistence"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns
        /persistence/persistence_1_0.xsd">
```

```
5   <persistence-unit name="PubSubEJBPU" transaction-type="JTA">
6     <provider>org.hibernate.ejb.HibernatePersistence</provider>
7     <jta-data-source>jdbc/PostgreSQL</jta-data-source>
8     <class>it.ditech.ejb.entity.Advertisement</class>
9     <class>it.ditech.ejb.entity.AssociationXpath</class>
10    <class>it.ditech.ejb.entity.Client</class>
11    <class>it.ditech.ejb.entity.Content</class>
12    <class>it.ditech.ejb.entity.Logging</class>
13    <class>it.ditech.ejb.entity.Namespace</class>
14    <class>it.ditech.ejb.entity.Publish</class>
15    <class>it.ditech.ejb.entity.Route</class>
16    <class>it.ditech.ejb.entity.Subscription</class>
17    <class>it.ditech.ejb.entity.SubscriptionXpath</class>
18    <class>it.ditech.ejb.entity.Trasformation</class>
19    <class>it.ditech.ejb.entity.Xpath</class>
20    <exclude-unlisted-classes>true</exclude-unlisted-classes>
21    <properties>
22      <property name="hibernate.connection.driver_class" value="org.postgresql.Driver"
          />
23      <property name="hibernate.connection.url" value="jdbc:postgresql://localhost
          :5432/jbi_database"/>
24      <property name="hibernate.connection.username" value="jbi_user"/>
25      <property name="hibernate.connection.password" value="password"/>
26      <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect
          "/>
27      <property name="hibernate.connection.pool_size" value="600"/>
28      <property name="hibernate.connection.autoReconnect" value="true"/>
29      <property name="hibernate.generate_statistics" value="false"/>
30      <property name="hibernate.show_sql" value="true"/>
31      <property name="hibernate.use_sql_comments" value="true"/>
32      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
33    </properties>
34  </persistence-unit>
35 </persistence>
```

Listing 4.7: `persistence.xml`

Come si può notare nella prima parte del file di configurazione sono state inserite tutti gli Entity Bean e viene impostato il data source jdbc/PostgreSQL.

Nella seconda parte del file vengono elencate tutte le proprietà di Hibernate utilizzate per la connessione al dbms e per la gestione delle tabelle; in particolare i driver di PostgreSQL da utilizzare e le informazioni riguardanti nome utente,

password e url di connessione.

4.3.2 Logica e Session Beans

In questa sezione andremo ad occuparci delle Session Bean che gestiscono la logica del sistema e l'accesso alla banca dati.

Le session bean utilizzate sono tre PubSubLoggingBean, PubSubStoringBean e PubSubForwardingBean. Ognuna di queste session implementa la sua interfaccia remota; rispettivamente : PubSubLoggingRemote, PubSubStoringRemote e PubSubForwardingRemote. Di seguito analizzeremo ognuna di queste session bean e spiegheremo la logica al suo interno.

Nella session PubSubLoggingBean, si procede a salvare il messaggio in arrivo sulla banca dati nella tabella logging. Questa operazione consente di creare uno storico dei messaggi ricevuti dal sistema con una serie di operazioni che consentono di caratterizzare la tipologia di messaggio, il time stamp della ricezione e l'associazione con il client che ha prodotto il messaggio stesso.

Questa operazione qualora terminasse con successo restituisce l'identificatore della tupla che rappresenta il messaggio nella base di conoscenza.

Una nota va aggiunta sulla modalità di salvataggio del messaggio stesso: nel data base non viene salvato direttamente il DOM dell'XML, ottenuto dopo la ricezione, ma la corrispondente stringa.

Infatti per avere delle prestazioni migliori abbiamo preferito trasformare il DOM, utilizzato per la manipolazione del messaggio, in una stringa (UTF-8) e salvarla sulla banca dati, come flusso di byte.

In questo modo abbiamo evitato le onerose operazioni di serializzazione della classe DOM di Apache.

Dal punto di vista del codice sorgente Java questa Session Bean non mostra nessun aspetto particolarmente interessante.

Nella PubSubStoringBean si procede al salvataggio strutturato del messaggio ricevuto. Per salvataggio strutturato si intende la suddivisione del messaggio nei concetti elementari che lo compongono e successivamente, il loro relativo salvataggio sul data base.

Per ogni messaggio, in base al tipo, viene creato un Entiy Bean apposito che può essere uno dei seguenti:

- **Advertisement**: questo oggetto permette di salvare un messaggio di tipo advertise e consente il collegamento con i concetti fondamentali del messaggio stesso. Nel caso dell'operazione di advertise siamo in presenza di un unico concetto fondamentale: il namespace.

  Al momento della creazione dell'oggetto Advertisement, viene estratto un entity Namespace che verrà poi collegato tramite il vincolo della relazione uno-a-molti all'oggetto Advertisement appena creato.

  Dopo che la relazione tra le due entità è stata stabilita, l'Advertisement è salvato nella banca dati.

- **Subscription**: questo oggetto è utilizzato per salvare i messaggi di tipo Subscribe.

  I concetti elementari contenuti in questo tipo di messaggio sono tre: il namespace, l'xpath e il valore. Quindi per ognuno di essi viene creato un oggetto rispettivamente Xpath, SubscriptionXpath e naturalmente Subscription.

  Questi oggetti collegati tra loro dalle relazioni che li accomunano vengono salvati sulla banca dati, prima del salvataggio viene estratto il Namespace corrispondente e associato al Subscription.

- **Publish**: l'oggetto Publish viene coinvolto nella fase di salvataggio dei messaggi di Notify che risulta essere la fase più complessa di tutte quante poiché coinvolge un grande numero di altri Entity Bean.

  Oltre al Namespace è coinvolto anche l'oggetto Content con il quale viene salvato il contenuto effettivo del Publish, l'oggetto Subscription, l'oggetto Xpath ed anche l'oggetto Route.

  L'Entity Bean Route viene utilizzato per creare una sorta di tabella di routing nella quale vengono associati destinatari e contenuto.

Nel caso dei messaggi di Publish viene eseguito una procedura detta di Replacement atta ad aggiornare eventuali versioni antecendenti del contenuto.

Nel listato seguente possiamo vedere il codice Java (semplificato) utilizzato per la fase di salvataggio di un messaggio di Publish.

```
1 private boolean storeNot(DOMDocument dom, Long Id) throws java.lang.Exception {
2         boolean replacement = false;
```

```
3         Publish pub = new Publish();
4         Logging log = getLoggerRecord(Id);
5         pub.setLoggingId(log);
6         Client client = log.getClientId();
7         pub.setClientId(client);
8         String url = null;
9         try {
10            url = EJBUtils.setUri(dom.getRootElement().element("NotificationMessage").element("
                  ProducerReference").element("Metadata").element("Address").getStringValue());
11        } catch (Exception ex) {
12            return false;
13        }
14        Namespace namespace = null;
15        try {
16            namespace = (Namespace) em.createQuery("FROM Namespace n WHERE n.uri LIKE (:uri)").
                  setParameter("uri", url).getSingleResult();
17        } catch (Exception ex) {
18            return false;
19        }
20        pub.setNamespaceId(namespace);
21        Content content = new Content();
22        DOMDocument contenuto = null;
23        try {
24            contenuto = EJBUtils.stringToDom(((Element) dom.selectNodes("/wsn:Notify/wsn:
                  NotificationMessage/wsn:Message/node()").get(1)).asXML());
25        } catch (Exception ex) {
26            return false;
27        }
28        content.setContent(contenuto.asXML().getBytes());
29        content.setNamespaceId(namespace);
30        pub.setContentId(content);
31        Xpath xpath = (Xpath) namespace.getReplacementXpathId();
32        ArrayList list = new ArrayList(xpath.getApplicationXpathList());
33        ApplicationXpath ax = null;
34        if (list.size() == 0) {
35            replacement = false;
36            try {
37                em.persist(content);
38                em.persist(pub);
39            } catch (Exception ex) {
40                return false;
41            }
42        } else {
43            Iterator iteApplication = list.listIterator();
44      while (iteApplication.hasNext()) {
45                ax = (ApplicationXpath) iteApplication.next();
46                if (executeXPath(xpath.getStatement(), ax.getValues(), contenuto)) {
47                    replacement = true;
48                    break;
49                } else {
50                    try {
51                        em.persist(content);
52                        em.persist(pub);
53                    } catch (Exception ex) {
54                        return false;
55                    }
56                }
57            }
58        }
59        Long id = content.getContentId();
60        if (replacement) {
61            id = doReplacement(ax, namespace, contenuto);
62            if (id == null) {
63                return false;
```

```
64              }
65          } else {
66              em.flush();
67              if (!insertApplicationXPath(pub, content, xpath)) {
68                  return false;
69              }
70          }
71          return forward(id);
72      }
```

Listing 4.8: `StoreNotify.java`

Nella prima parte del codice fino alla riga 30 vengono creati gli Entity Bean necessari per l'archiviazione del messaggio ed in particolare un oggetto Publish nel quale verranno salvate appositamente le informazioni necessarie, un oggetto Namespace che viene recuperato dalla banca dati e un oggetto Content che servirà a salvare il contenuto effettivo del messaggio di Publish.

Dopo questa prima fase inizia il controllo per il Replacement. A partire dal namespace del contenuto viene recuperato un oggetto di tipo ReplacementXpath.

Il ReplacementXpath consiste in un XPath associato al namespace che deve essere eseguito sul contenuto del messaggio.

Associato all'oggetto ReplacementXpath esiste anche una lista di valori che può essere vuota o meno. Se la lista è non vuota i risultati ottenuti vengono confrontati con quelli presenti, nel caso in cui la lista sia vuota l'operazione di Replacement termina.

Se i risultati ottenuti sono diversi da quelli presenti nella lista l'operazione termina; se invece i valori coincidono si procede ad aggiornare il contenuto presente sul data base con quello appena ricevuto.

In altri termini attraverso l'XPath di replacement si procede a controllare che il contenuto che si sta processando sia una nuova versione dei contenuti già presenti sulla banca dati, se così è i contenuti salvati in precedenza vengono aggiornati.

Questa parte di procedura viene eseguita nelle righe che vanno dalla numero 31 alla riga numero 70. Le procedure richiamate sono per il controllo e l'esecuzione dell'XPath (riga 46), la procedura di replacement alla riga 60 ed infine l'inserimento dei valori dell'XPath appena eseguito nella banca dati, (riga 67).

Quest'ultima operazione viene eseguita solamente nel caso in cui il replacement non sia necessario.

La procedura forward della riga 71 è molto importante e serve a creare l'associazione tra client e contenuto. Dopo aver recuperato le sottoscrizioni per il namespace sul contenuto vengono eseguiti gli Xpath presenti nelle sottoscrizioni.

Se i risultati ottenuti coincidono con quelli specificati viene instanziato un nuovo oggetto Route, per creare la sopracitata associaziaone tra Subscriber e Content.

Il flag `routed` viene settato a false per indicare alla session di inoltro che il messaggio ancora non è stato inviato al client.

La procedura di forwarding viene richiamata anche nella procedura di Replacement, per preparare all'inoltro i contenuti appena aggiornati.

L'ultima Session Bean è la PubSubForwardBean che ha la funzione di andare a recuperare gli oggetti di tipo Route e controllare se sono da inoltrare oppure no.

Questo controllo viene fatto consultando il flag dell'Entity Bean e se è settato a `false` il contenuto va inoltrato al rispettivo client. Al termine il flag va posto a `true`.

Questa operazione viene svolta ovviamente solo nel caso in cui il messaggio che ha attivato la catena di Service Engine è di tipo Publish.

Nel caso in cui il messaggio è di un tipo diverso la computazione termina dopo l'esecuzione del servizio del secondo engine, quindi la session PubSubForwardBean non viene richiamata.

Capitolo 5

Test e Performance

Introduzione

In questo capitolo tratteremo la fase di testing e i risultati ottenuti. In particolare abbiamo scelto di testare a fondo l'operazione di Publish in quanto operazione maggiormente invocata rispetto alle altre, in una tipica modalità di utilizzo del servizio. Il capitolo è stato suddiviso in quattro sezioni, dove si parlerà, rispettivamente, della configurazione dell'ambiente di test e dei test effettuati per accertare le performance e la robustezza del sistema.

5.1 Configurazione dei Test

I test sono stati eseguiti su un host Dell modello Optiplex SX280 dotato di due processori Intel Pentium 4 a 3.0 GHz e di 1GB di memoria RAM; il sistema operativo montato è Ubuntu versione 9.04 con kernel 2.6.28-15-generic.

La macchina virtuale Java utilizzata per i test è la versione 1.6.0_16 di Sun Microsystem, a cui erano riservati 768 MB di memoria RAM.

L'application server utilizzato è Glassfish nella versione 2.1 con OpenESB integrato; l'application server è configurato per mantenere un massimo di 600 connessioni alla banca dati e per gestire le connessioni HTTP secondo i parametri elencati sotto:

```
1 server.http-service.connection-pool.max-pending-count = 4096
2 server.http-service.connection-pool.queue-size-in-bytes = 4096
```

```
server.http-service.connection-pool.receive-buffer-size-in-bytes = 4096
server.http-service.connection-pool.send-buffer-size-in-bytes = 8192

server.http-service.keep-alive.max-connections = 250
server.http-service.keep-alive.thread-count = 1
server.http-service.keep-alive.timeout-in-seconds = 30

server.http-service.keep-alive.countconnections-count = 1869
server.http-service.keep-alive.countflushes-count = 0
server.http-service.keep-alive.counthits-count = 359873
server.http-service.keep-alive.countrefusals-count = 1428
server.http-service.keep-alive.counttimeouts-count = 0
server.http-service.keep-alive.maxconnections-count = 250
server.http-service.keep-alive.secondstimeouts-count = 30
```

Listing 5.1: `Configurazione parametri HTTP`

Per effettuare i test e prendere le misurazioni abbiamo utilizzato SoapUI in quanto JMeter non consente di invocare servizi web che fanno uso di attachments mime. SoapUI è un tool che consente, a partire da un file WSDL, di definire e implementare dei test su WebServices che fanno uso del protocollo SOAP via HTTP.

I test accertano le performance del sistema a fronte della ricezione di messaggi di tipo publish, in quanto l'operazione publish è l'operazione rilevante, sia come numero di invocazioni che come carico computazionale, per quanto riguarda i sistemi di content notification. Ai fini di questo test abbiamo definito un publisher e un numero di subscriber che varia in base al tipo di test effettuato. I contenuti inviati si possono ripetere al fine di scatenare anche operazioni di aggiornamento delle versioni dei contenuti.

Per ogni invio di contenuti, SOAPUI colleziona il tempo dell'evento, il tempo medio di risposta, le transazioni per secondo e la dimensione del flusso di dati in termini di Kb/s.

5.2 Preparazione all'attività di testing

In genere, indipendentemente dal linguaggio di programmazione adottato, le attività di sviluppo si concentrano sulla corretta formalizzazione delle logiche fun-

zionali e non funzionali del servizio tuttavia non garantiscono che il prodotto sia in grado di supportare, con continuità di servizio, le esigenze di un ambiente di produzione.

Inoltre vale la pena ricordare che il prototipo di questa tesi è stato scritto in Java. Tale linguaggo ha una curva di apprendimento e di messa in pratica molto vantaggiosa in quanto automatizza la gestione di aspetti di sviluppo, come ad esempio l'uso della memoria, le cui gestioni attraverso linguaggi di programmazione meno evoluti sarebbero a carico dello sviluppatore. Tuttavia, la Java Virtual Machine (JVM) non è in grado di garantire che le risorse di cui dispone siano utilizzate nella maniera più appropriata; molto dipende dallo stile e dagli accorgimenti adottati dallo sviluppatore nella definizione del codice. Riferimenti pendenti (*danglingreferences*) e stream non chiusi, che provocano buffer non deallocati, sono le cause spesso trascurate di un precoce esaurimento della memoria e di una maggior frequenza di attivazione del garbage collector; nella sua esecuzione, tale componente è bloccante rispetto alle attività della JVM e inefficace nella deallocazione della memoria intrappolata dalle cause sopra menzionate.

Per valutare e correggere la qualità della gestione della memoria nel prototipo sviluppato, al termine degli sviluppi abbiamo dedicato la prima sessione di testing alla misurazione dell'uso della memoria a runtime impiegando uno strumento di profilazione chiamato YourKit. In particolare, abbiamo sviluppato un semplice client che inviasse richieste di publish al servizio e abbiamo sfruttato YourKit per monitorare l'uso della memoria, in termine di istanze di classi java allocate dalla JVM e di quantità di memoria RAM occupata da tali istanze.

Attraverso diverse iterazioni di questo test, siamo riusciti a individuare tutte le cause della mancata deallocazione di memoria da parte del garbage collector. Al termine di ogni iterazione abbiamo fatto una retroazione sull'implementazione per risolvere le cause evidenziate garantendo al prototipo una maggiore robustezza a fronte di una continuità di richieste di servizio; alcune di tali cause erano dovute allo stile di programmazione del prototipo stesso e altre erano dovute alla modalità d'uso di alcune librerie che non mancavano di chiarire, nella loro documentazione, la corretta semantica di utilizzo dei componenti da esse offerti.

Questa sessione di pre-testing è terminata quando, all'ultima iterazione, abbiamo accertato l'efficacia del garbage collector nel ripristinare, a seguito di ogni sua

esecuzione, la quantità di memoria libera individuata alla partenza del servizio.

5.3 Andamento del throughput al variare delle dimensioni del messaggio

In questa sezione presentiamo i test per la determinazione dell'andamento del throughput all'aumentare della dimensione del contenuto pubblicato. La configurazione di test prevede lo schieramento del servizio, di un publisher, e di un subscriber. I test sono mirati ad accertare il throughput del sistema; abbiamo quindi approssimato il massimo carico che consenta al sistema di rimanere in esecuzione senza che le performance degenerino col passare del tempo. I test eseguiti sono stati in tutto cinque e sono stati realizzati aumentando progressivamente la dimensione del contenuto pubblicato: 1.66 Kb, 3 Kb, 5 Kb, 8 Kb e 10Kb. Ogni test effettuato ha la durata di due ore.

Per ogni dimensione di messaggio, abbiamo fatto un pre-test per stimare la frequenza di invio tale da occupare il servizio senza saturarlo consentendo quindi una performance di servizio che non degenera nel tempo. I grafici nelle figure 5.1, 5.2, 5.3, 5.4 e 5.5 dimostrano l'andamento del throughput per due ore quando il server

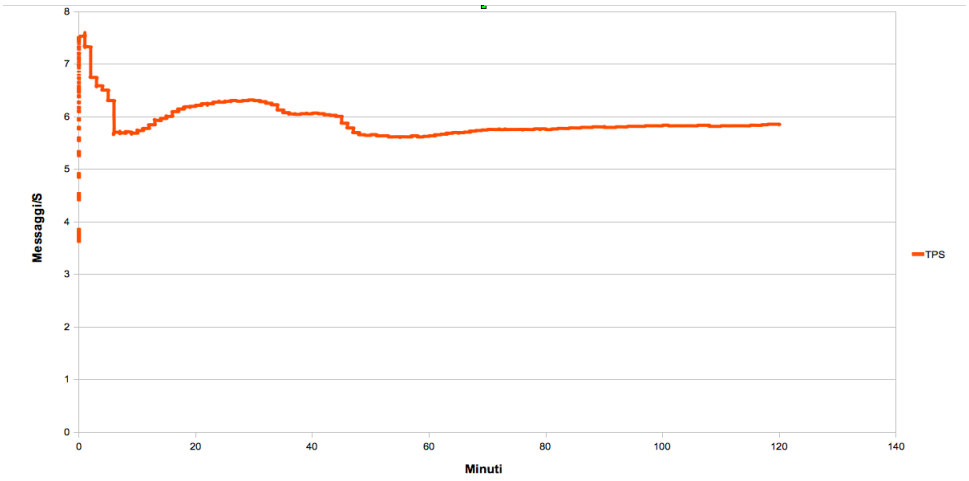

Figura 5.1: *Andamento del throughput con contenuti di 1.66 Kb*

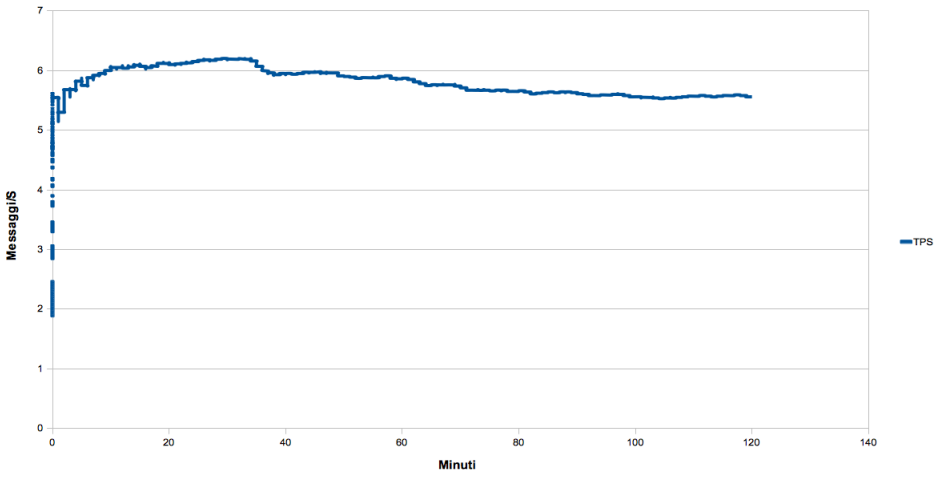

Figura 5.2: *Andamento del throughput con contenuti di 3 Kb*

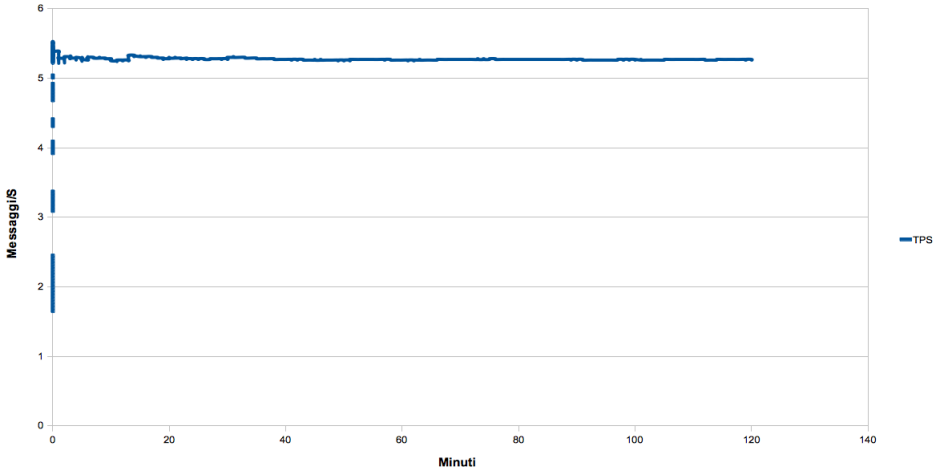

Figura 5.3: *Andamento del throughput con contenuti di 5 Kb*

è sollecitato rispettivamente con 7, 6, 6, 6 e 5 publish al secondo. I grafici riportano sull'asse delle ascisse il tempo in minuti, mentre sulle ordinate viene riportato il valore del throughput. Come è visibile direttamente, tutti i test presentano lo stesso andamento; dopo un periodo di assestamento, il throughput assume un valore pressoché costante che si mantiene per tutta la durata del test. Fa una leggera eccezione il primo test per il quale non siamo riusciti ad approssimare esattamente la

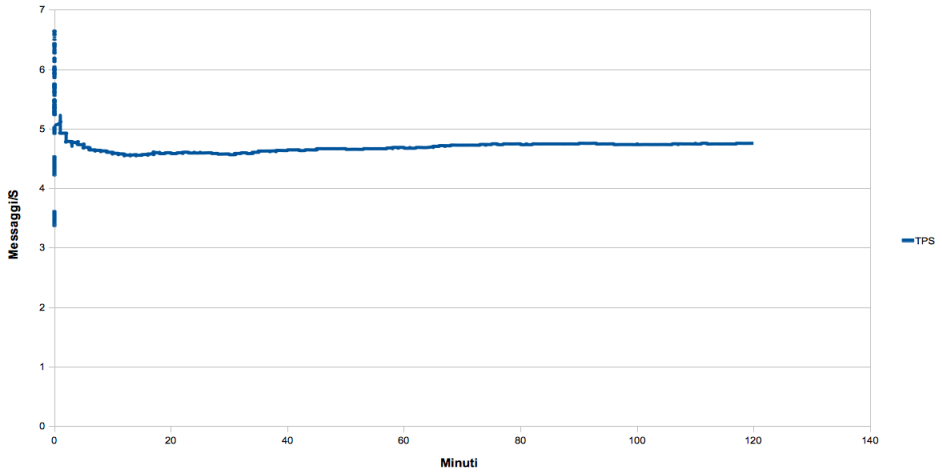

Figura 5.4: *Andamento del throughput con contenuti di 8 Kb*

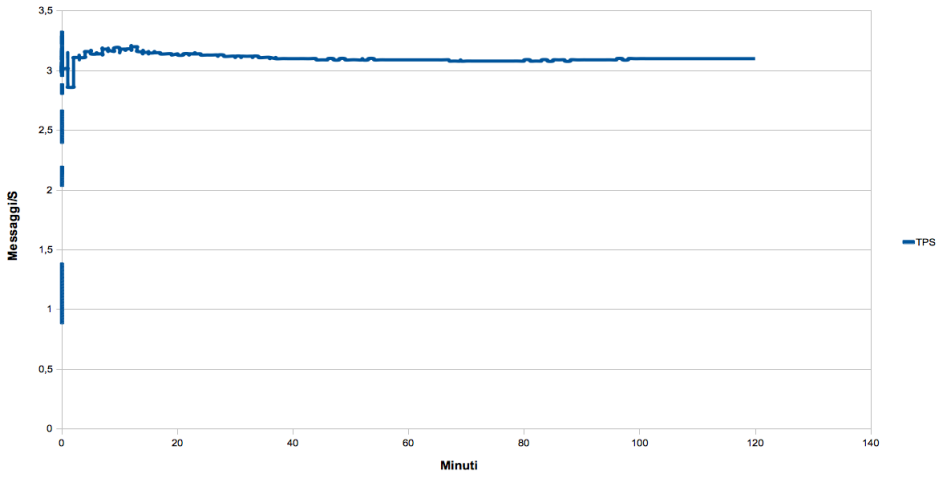

Figura 5.5: *Andamento del throughput con contenuti di 10 Kb*

frequenza di invio; questo è dimostrato dall'andamento di crescita del throughput rilevato attraverso SoapUI. In questo caso possiamo dire di aver raggiunto un'approssimazione per difetto. Vale la pena ricordare che a fronte di una publish tutte le operazioni di logging, storing e forwarding sono eseguite in maniera sincrona prima di restituire un risultato all'applicazione cliente. Questo ci fa capire che, a seconda della dimensione del messaggio, per un contenuto di dimensione variabile

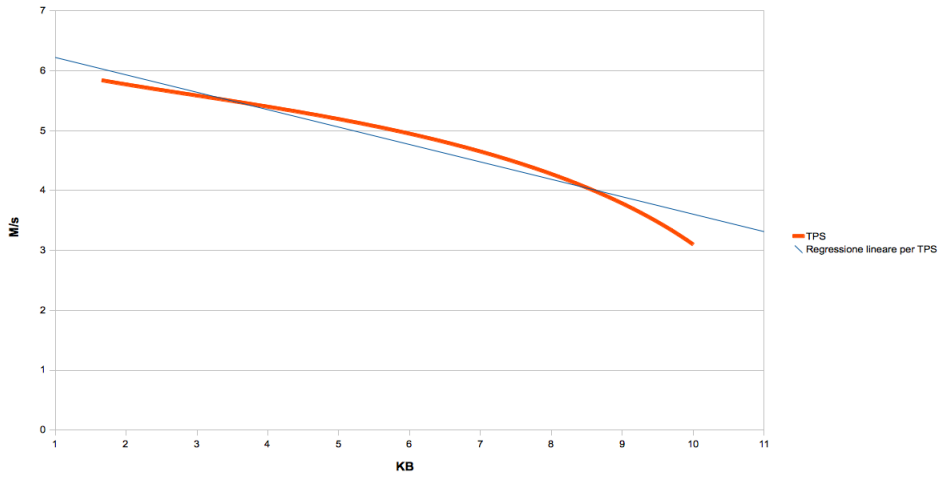

Figura 5.6: *Andamento del throughput al variare delle dimensioni dei messaggi*

tra 1,66KB e 10KB, riusciamo ad implementare tutte le funzionalità sopra citate in un tempo medio che varia tra 170 e 300 millisecondi circa, che trova giustificazione nel numero di accessi alla banca dati e nel numero delle operazioni svolte per gestire la pubblicazione e l'inoltro dei contenuti. Nella figura 5.6 i risultati dei test precedenti sono comparati al fine di descrivere l'andamento del throughput del sistema al variare della dimensione dei contenuti pubblicati. Il grafico mostra come vi sia un rapporto di proporzionalità inversa tra la dimensione del messaggio e il throughput del sistema. Fino a 8KB di dimensione del contenuto, la proporzionalità inversa è di ordine lineare; oltre gli 8KB, la relazione di proporzionalità inversa sembra assumere un andamento sovralineare. Questo dimostra che le peformance del sistema scalano al variare della dimensione del contenuto nello spettro di dimensioni considerato.

5.4 Andamento del throughput al variare dei subscriber

In questa sezione presentiamo i test per la determinazione dell'andamento del throughput all'aumentare del numero di sottoscrittori. La configurazione di test

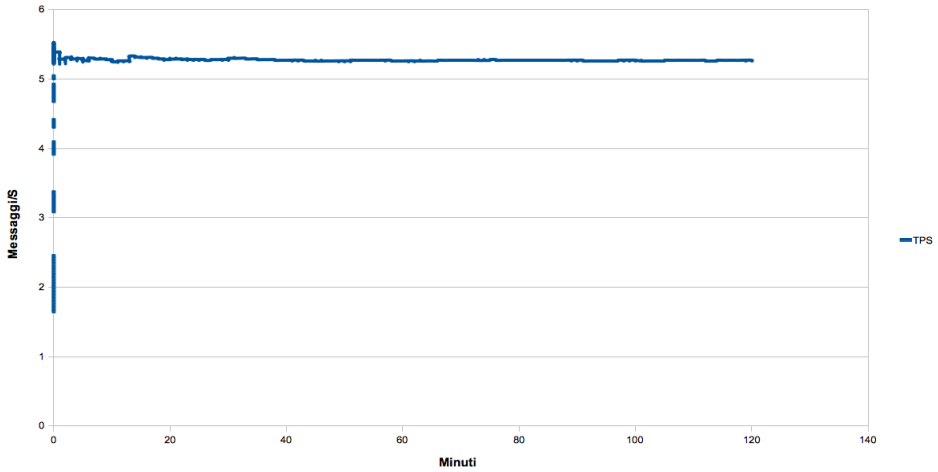

Figura 5.7: *Andamento del throughput con 1 subscriber*

prevede lo schieramento del servizio, di un publisher, e di un numero di subscriber variabile tra 1 e 16, con una dimensione del contenuto trasmesso uguale a 5 KB. I test sono mirati ad accertare il throughput del sistema; abbiamo quindi approssimato il massimo carico che consenta al sistema di rimanere in esecuzione senza che le performance degenerino col passare del tempo. I test eseguiti sono stati in tutto cinque e sono stati realizzati aumentando progressivamente il numero di sottoscrittori per ogni contenuto pubblicato: 1,2,4,8,16. Ogni test effettuato ha la durata di due ore.

Per ogni dimensione di messaggio, abbiamo fatto un pre-test per stimare la frequenza di invio tale da occupare il servizio senza saturarlo consentendo quindi una performance di servizio che non degenera nel tempo. I grafici nelle figure 5.7, 5.8, 5.9, 5.10 e 5.11 dimostrano l'andamento del throughput per due ore quando il server è sollecitato rispettivamente con 6, 3, 3, 2 e 2 publish al secondo. I grafici riportano sull'asse delle ascisse il tempo in minuti, mentre sulle ordinate viene riportato il valore del throughput. Come è visibile direttamente, tutti i test presentano lo stesso andamento; dopo un periodo di assestamento, il throughput assume un valore pressoché costante che si mantiene per tutta la durata del test.

Vale la pena ricordare che a fronte di una publish tutte le operazioni di logging, storing e forwarding sono eseguite in maniera sincrona prima di restituire un risul-

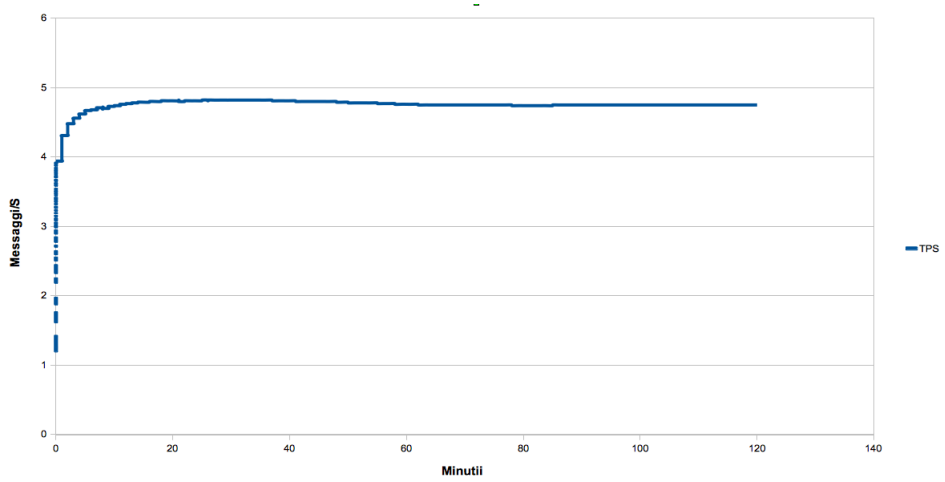

Figura 5.8: *Andamento del throughput con 2 subscriber*

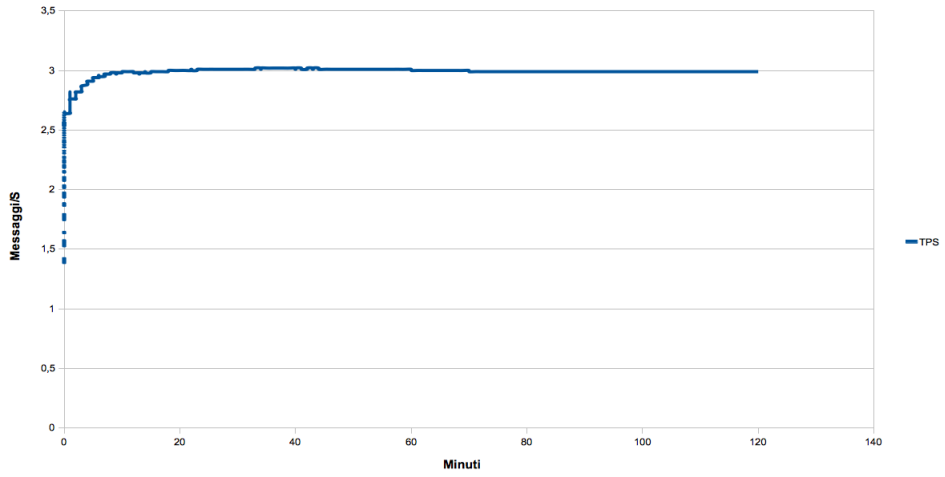

Figura 5.9: *Andamento del throughput con 4 subscriber*

tato all'applicazione cliente. Questo ci fa capire che, a seconda della dimensione del messaggio, per numero di subscriber che varia da 1 a 16, riusciamo ad implementare tutte le funzionalità sopra citate in un tempo medio che varia tra 190 e 800 millisecondi circa, che trova giustificazione nel numero di accessi alla banca dati e nel numero delle operazioni svolte per gestire la pubblicazione e l'inoltro dei contenuti.

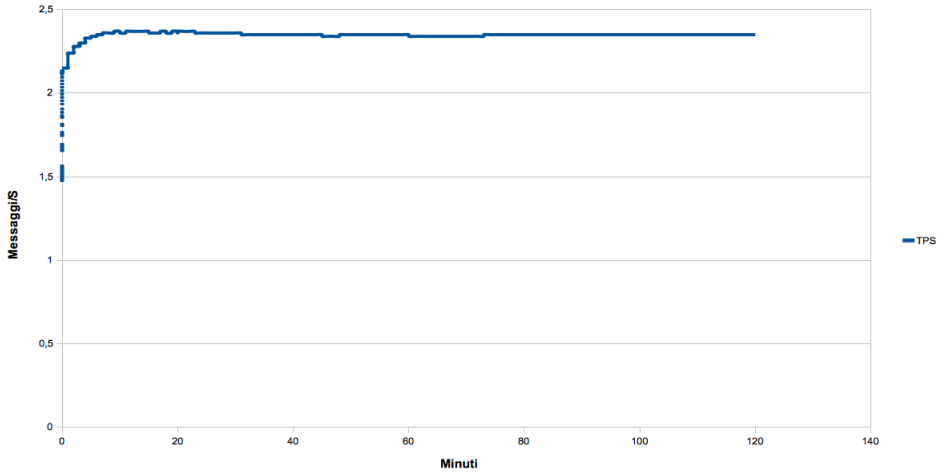

Figura 5.10: *Andamento del throughput con 8 subscriber*

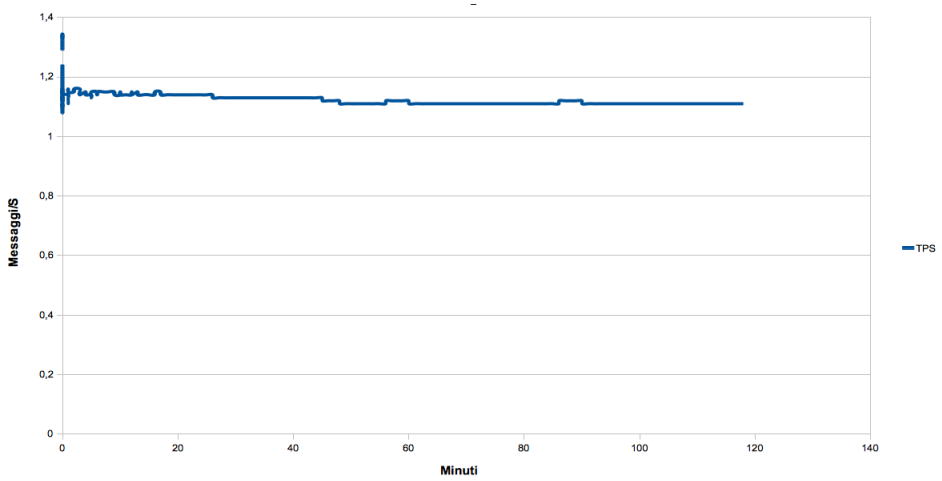

Figura 5.11: *Andamento del throughput con 16 subscriber*

Nella figura 5.12 i risultati dei test precedenti sono comparati al fine di descrivere l'andamento del throughput del sistema al variare del numero di subscriber. Il grafico mostra come vi sia un rapporto di proporzionalità inversa di ordine lineare tra la dimensione del messaggio e il throughput del sistema. Questo dimostra che le peformance del sistema scalano al variare della dimensione del contenuto nello spettro di dimensioni considerato, se si considera che tutte le attività di inoltro

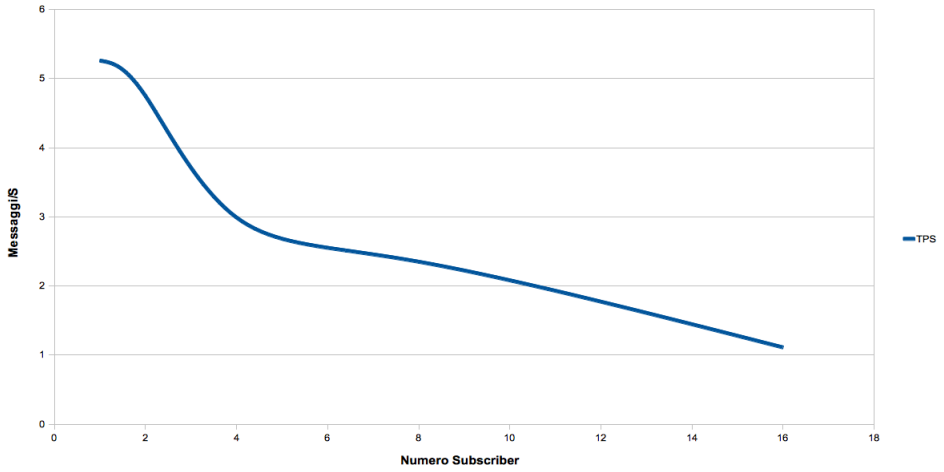

Figura 5.12: *Andamento del throughput al variare del numero di subscriber*

vengono svolte in maniera sincrona rispetto alla publish.

5.5 Andamento del throughput al variare del carico, in condizioni di stress

In questa sezione presentiamo i test per la determinazione dell'andamento del throughput all'aumentare del carico di publish, in condizioni di stress. La configurazione di test prevede lo schieramento del servizio, di un publisher, e un subscriber, con una dimensione del contenuto trasmesso uguale a 1,66 KB. Ricordiamo che il carico ottimale per la dimensione del contenuto impiegata, calcolato nel primo esperimento, si trova intorno alle 7 publish al secondo. I test sono mirati ad accertare il throughput del sistema in condizioni di stress che fanno degenerare le prestazioni del servizio col passare del tempo. I test eseguiti sono stati in tutto cinque e sono stati realizzati aumentando progressivamente il carico di lavoro richiesto al server: 144%, 200%, 244%, 300%, 333%, ovvero generando rispettivamente 10, 14, 17, 21 e 23 publish al secondo. Ogni test effettuato ha la durata di tre ore.

I grafici nelle figure 5.13, 5.14, 5.15, 5.16 e 5.18 dimostrano l'andamento del throughput per tre ore quando il server è sollecitato rispettivamente con 10, 14, 17, 21 e 23 publish al secondo. I grafici riportano sull'asse delle ascisse il tempo

in minuti, mentre sulle ordinate viene riportato il valore del throughput. Come è visibile direttamente, tutti i test presentano lo stesso andamento; dopo un periodo di assestamento, il throughput inizia a decrescere ad una velocità che si mantiene quasi costante per tutta la durata del test.

Un aspetto interessante che vale la pena osservare per tutti i test effettuati è come il server, dopo un carico che arriva ad essere tre volte maggiore rispetto a

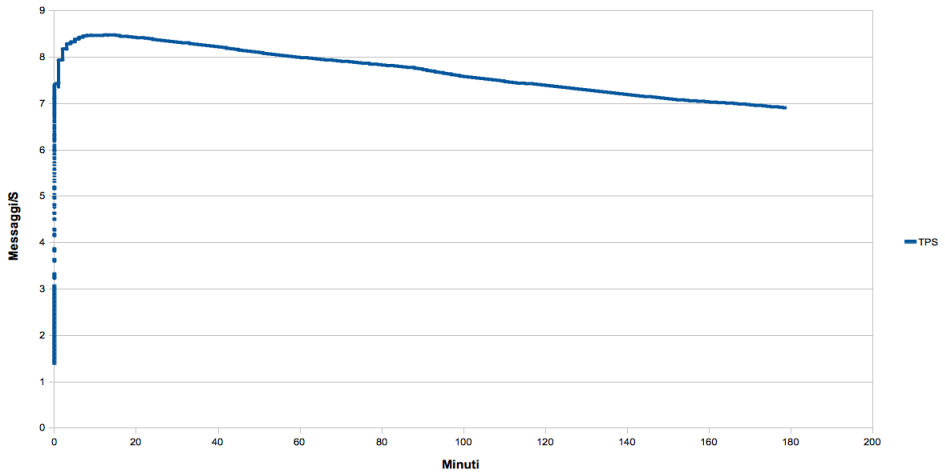

Figura 5.13: *Andamento del throughput con 144% del carico*

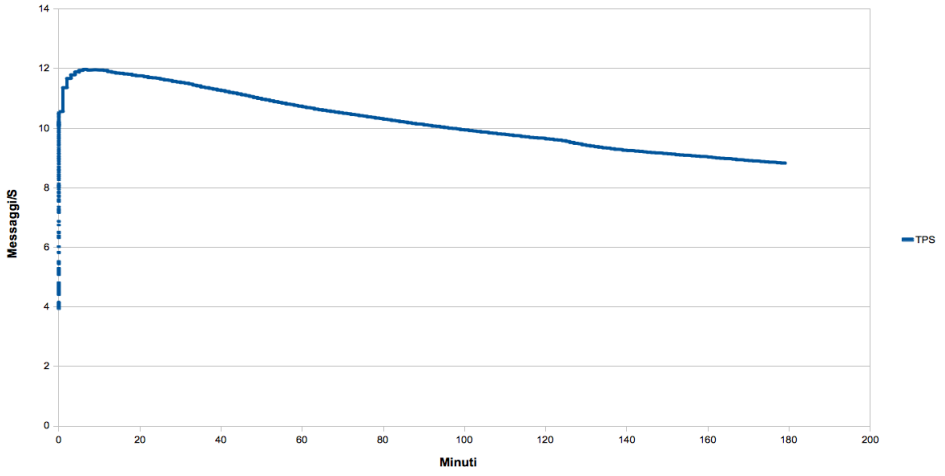

Figura 5.14: *Andamento del throughput con 200% del carico*

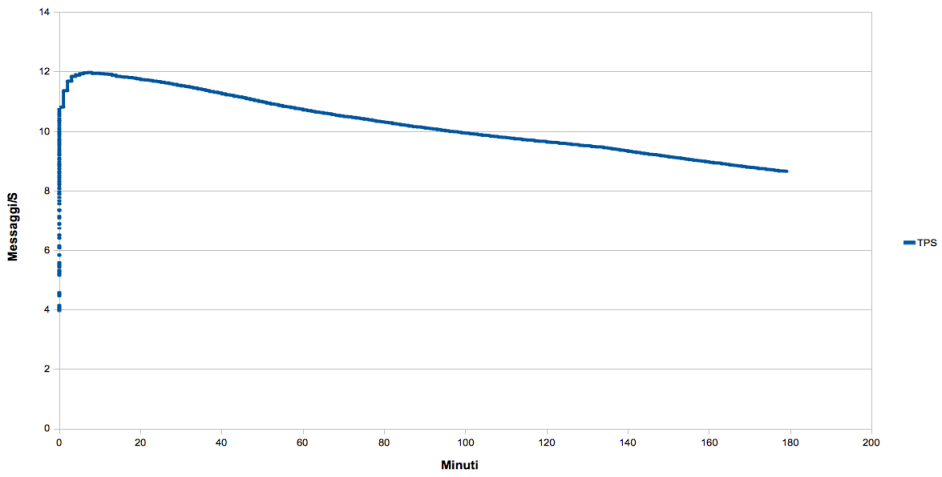

Figura 5.15: *Andamento del throughput con 244% del carico*

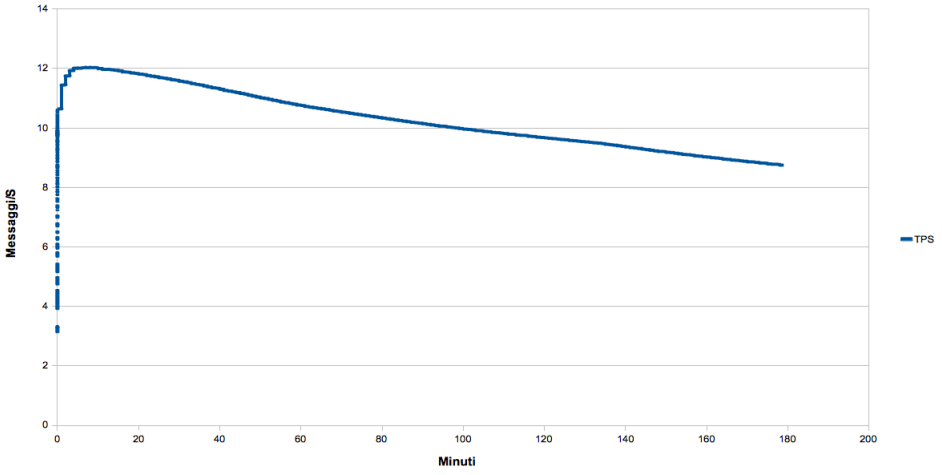

Figura 5.16: *Andamento del throughput con 300% del carico*

quello ottimale, riesca in tutti i casi di test effettuati a non subire un crash dopo tre ore di sollecitazione. Questo risultato fa credere che il sistema sia robusto a sufficienza per sopportare eventuali picchi di carico e dà una dimostrazione dell'efficacia dell'ottimizzazione operata sulla gestione della memoria guidata dalle attività di testing descritte all'inizio di questo capitolo.

Per ogni test effettuato, l'andamento peggiorativo delle performance ci permet-

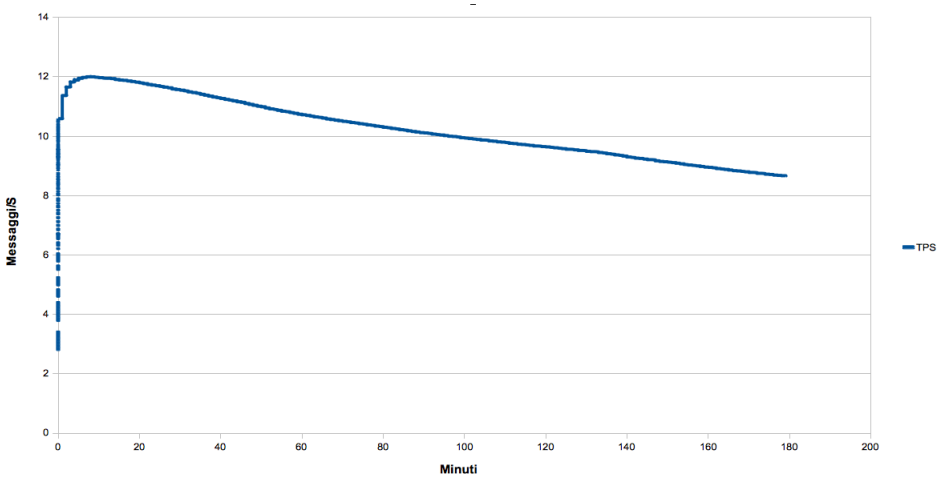

Figura 5.17: *Andamento del throughput con 333% del carico*

te di calcolare un tempo teorico di crash che definiamo corrispondente al tempo necessario al sistema per passare dal picco massimo di performance fino a raggiungere un throughput pari a 0, mantenendo invariato lo stress indotto dal ritmo di publish. Nella realtà, il sistema dovrebbe subire un crash in coincidenza dell'esaurimento delle risorse dedicate ad accogliere le richieste che non riescono ancora ad essere servite; tuttavia riteniamo che il tempo teorico di crash possa essere un buon indice della robustezza del sistema nello specifico caso di test.

La figura 5.18 mostra l'andamento del tempo teorico di crash al variare del carico esercitato sul sistema. Qui possiamo notare che il sistema ha un tempo teorico di crash di circa 13 ore con un carico del 133% del carico ottimale e che non scende sotto le 10 ore con una sollecitazione pari al 333% dello stesso carico; dall'esperienza che abbiamo maturato testando questo sistema, abbiamo ragione di credere che in uno scenario reale questo risultato corrisponda ad un time to crash minimo compreso tra le 5 e le 6 ore. Questo risultato dimostra la robustezza del sistema che abbiamo implementato.

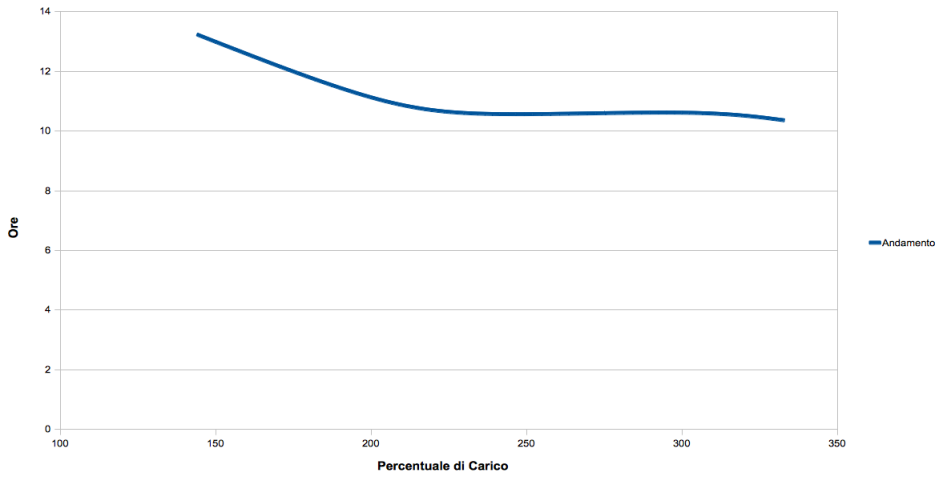

Figura 5.18: *Time to crash al variare del carico*

Conclusioni e Sviluppi Futuri

In questa tesi abbiamo descritto il nostro lavoro di studio, progettazione, implementazione e testing di un sistema di comunicazione di tipo Publish/Subscribe con routing content-based. In particolare, siamo riusciti a portare a termine il nostro lavoro di tesi soddisfacendo tutti i requisiti di multicanalità, flessibilità, persistenza, performance e robustezza che ci eravamo posti come obiettivo:

- **Multucanalità**: L'adozione della tecnologia JBI rende possibile il disaccoppiamento delle attività di sviluppo delle logiche di business dalle attività di sviluppo delle interfacce ai protocolli di trasporto. Attualmente abbiamo sviluppato due livelli di trasporto, uno SOAP/HTTP e uno tramite file system. È tuttavia possibile con uno sforzo limitato e senza richiedere cambiamenti al codice preesistente adattare il sistema a comunicare sfruttando un nuovo livello di trasporto;

- **Scalabilità**: I test effettuati dimostrano la scalabilità del sistema, in termini di performance, al variare della dimensione del contenuto pubblicato e del numero di subscriber registrati;

- **Flessibilità**:

  - L'adozione della tecnologia openESB rende possibile la veloce integrazione di componenti che implementano logiche accessorie (p.e., content syndication);

  - L'adozione di contenuti basati su XML che devono essere qualificati con il namespace di appartenenza e l'uso congiunto del linguaggio di interrogazione xpath consentono al sistema di gestire simultaneamente

tanti diversi formati di contenuto e di esprimere dettagliati e sofisticati criteri di sottoscrizione e instradamento;

- **Persistenza**:
  - Il componente di logging consente di mantenere le informazioni storiche di tutte le interazioni tra gli attori esterni e il sistema;
  - Il componente di storing mantiene una base di conoscenza dove, per ogni contenuto distinto, viene mantenuta la versione più recente;
- **Performance**: L'accertamento delle performance ha dimostrato come il sistema sia capace di supportare molteplici richieste di servizio simultaneamente e a propagare i contenuti verso molteplici subscriber. Vale la pena notare che tutti i test sono stati effettuati imponendo ai componenti di servizio di interagire in modalità sincrona con lo scopo di avere delle misurazioni che comprendessero tutte le fasi di elaborazione e di instradamento dei messaggi; tuttavia, è altresì da notare che tale modalità di interazione ha un impatto peggiorativo sulle performance misurate. Nell'ottica della messa in produzione del servizio, è ragionevole assumere che lo schieramento con modello di interazione asincrono migliori ulteriormente il livello di performance erogato;
- **Robustezza**:
  - i test che abbiamo svolto dimostrano la capacità del sistema implementato di tollerare per un tempo prolungato un carico di stress molto superiore al carico ottimale (il carico che il sistema è in grado di mantenere col miglior livello di performance) senza subire guasti di tipo crash;
  - attraverso la combinazione di tecniche di trasporto dei contenuti via attachments mime, di limitazione alla dimensione massima dei contenuti e di limitazione al numero di thread che possono operare contemporaneamente, abbiamo progettato una tecnica per prevenire l'efficacia di attacchi di tipo Denial of Service.

Bibliografia

[ACW01] Jing Deng Antonio Carzanica and Alexander L. Wolf. Fast forwarding for content-based networking. *Technical Report CU-CS-922-01*, 2001.

[BCSS99] Guruduth Banavar, Tushar Ch, Robert Strom, and Daniel Sturman. A case for message oriented middleware. In *In Proceedings of the 13th International Symposium on Distributed Computing*, pages 1–18. Springer-Verlag, 1999.

[BRC08] Stephen Buxton, Michael Rys, and Pat Case. XQuery and XPath full text 1.0 requirements. W3C working draft, W3C, May 2008. http://www.w3.org/TR/2008/WD-xpath-full-text-10-requirements-20080516/.

[BRWF01] Allen Brown, Jonathan Robie, Philip Wadler, and Matthew Fuchs. XML schema: Formal description. W3C working draft, W3C, September 2001. http://www.w3.org/TR/2001/WD-xmlschema-formal-20010925/.

[BTT+09] Tim Bray, Richard Tobin, Henry S. Thompson, Dave Hollander, and Andrew Layman. Namespaces in XML 1.0 (third edition). W3C recommendation, W3C, December 2009. http://www.w3.org/TR/2009/REC-xml-names-20091208/.

[Car98] A. Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks*. PhD thesis, Politecnico di Milano, 1998.

[CJT01] Luis Felipe Cabrera, Michael B. Jones, and Marvin Theimer. Herald: Achieving a global event notification service. In *HOTOS '01: Pro-*

ceedings of the Eighth Workshop on Hot Topics in Operating Systems, page 87, Washington, DC, USA, 2001. IEEE Computer Society.

[CNF98] G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. *Software Engineering, International Conference on*, 0:261, 1998.

[CW02] Antonio Carzaniga and Alexander L. Wolf. *Content-Based Networking: A New Communication Infrastructure*, volume 2538/2002. Springer Berlin / Heidelberg, New York, NY, USA, 2002.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[Erl05] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[JS08] Frank Jennings and David Salter. *Building SOA-Based Composite Applications Using NetBeans IDE 6*. Packt Publishing, 2008.

[PB02] Peter R. Pietzuch and Jean Bacon. Hermes: A distributed event-based middleware architecture. In *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 611–618, Washington, DC, USA, 2002. IEEE Computer Society.

[PSMB98] Jean Paoli, C. M. Sperberg-McQueen, and Tim Bray. XML 1.0 recommendation. first edition of a recommendation, W3C, February 1998. http://www.w3.org/TR/1998/REC-xml-19980210.

[SGM06] David Hull Steve Graham and Bryan Murray. Web services base notification 1.3. Technical report, Web Services Base Notification 1.3, Ottobre 2006.

[SHLP05] M.-T. Schmidt, B. Hutchison, P. Lambros, and R. Phippen. The enterprise service bus: making service-oriented architecture real. *IBM Syst. J.*, 44(4):781–797, 2005.

[Vin05] Steve Vinoski. Java business integration. *IEEE Internet Computing*, 9(4):89–91, 2005.

[ZS01] Yuanyuan Zhao and Rob Strom. Exploitng event stream interpretation in publish-subscribe systems. In *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 219–228, New York, NY, USA, 2001. ACM.